(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
23 August 2001 (23.08.2001)

PCT

(10) International Publication Number
WO 01/61475 A1

(51) International Patent Classification⁷: G06F 9/30, 9/318

(21) International Application Number: PCT/US01/04743

(22) International Filing Date: 13 February 2001 (13.02.2001)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
09/503,779 14 February 2000 (14.02.2000) US

(71) Applicant: CHICORY SYSTEMS, INC. [US/US]; 12365 Riata Trace Parkway, Building II, Suite 150, Austin, TX 78727 (US).

(72) Inventors: DERRICK, John, E.; 2424 Falcon Drive, Round Rock, TX 78681 (US). MCDONALD, Robert, G.; 6280 McNeil Drive, Apartment 507, Austin, TX 78729 (US).

(74) Agent: HOOD, Jeffrey, C.; Conley, Rose & Tayon, P.C., P.O. Box 398, Austin, TX 78767-0398 (US).

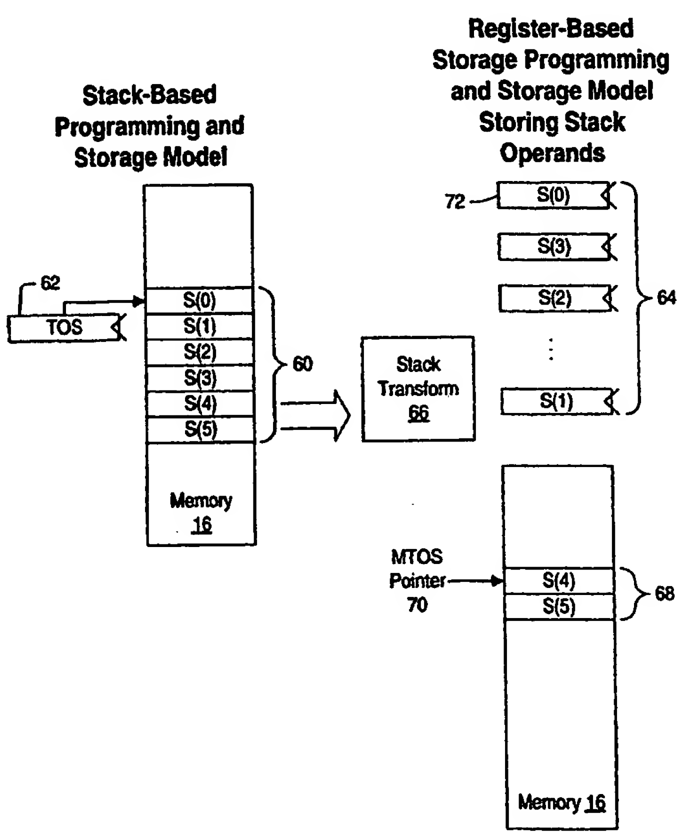
(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CR, CU, CZ, DE, DK, DM, DZ, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, UZ, VN, YU, ZA, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published:
— *with international search report*

[Continued on next page]

(54) Title: TRANSFORMING A STACK-BASED CODE SEQUENCE TO A REGISTER BASED CODE SEQUENCE



(57) Abstract: A system includes a CPU and a code translator. The CPU may execute instructions from a first instruction set, and the code translator may translate Java code sequences from Java bytecodes to code sequences having instructions defined in the first instruction set. The translated code sequences may be executed by the CPU. The Java instruction set is a stack-based instruction set, and the first instruction set may be a register-based instruction set. Accordingly, the code translator may translate stack operand references to register operand references. More particularly, the code translator may include a stack transform storage which stores a mapping of register indexes to stack items forming the top of the stack. The code translator may assign register indexes to a particular instruction in the translated code sequence according to the mapping and any changes to the mapping performed by other instructions which are concurrently translated and prior to the particular instruction. Accordingly, instructions in the translated code sequence may use register operands and thus operand access may be efficient when the translated code sequence is executed on the CPU.

WO 01/61475 A1

— *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

TITLE: TRANSFORMING A STACK-BASED CODE SEQUENCE TO A REGISTER BASED CODE SEQUENCE

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates to the field of programmable computing systems and, more particularly, to translation between instruction sets in computing systems.

2. Description of the Related Art

Java programs have become quite popular in recent years, particularly in view of the popularity of the Internet. A Java program is a program written to the Java language specification, and executes on a Java virtual machine (JVM). A JVM is an abstract computing machine which may be supported on any hardware platform (employing any suitable operating system and a native instruction set defined via any of a variety of architectures, e.g. x86, PowerPC, ARM, Alpha, etc.), and thus a Java program may execute on a variety of different hardware platforms. Thus, a Java program may be written and made available for download on the Internet, and the Java program may be executed on any hardware platform which supports the JVM.

In many cases, the JVM is itself a program written in the native instruction set of a given hardware platform. The JVM is called when a Java program is to be executed, and the JVM reads the instructions (termed "bytecodes") in the Java program one at a time in program order and emulates the execution behavior of the instructions on the hardware platform. Executing a program by having an interpreter program read each instruction and emulate that instruction's execution behavior is referred to as "interpreting" the program, or operating in an "interpreter mode".

Unfortunately, executing programs in an interpreter mode typically results in a slow execution speed. In an attempt to speed the execution, software just-in-time (JIT) compilers have been proposed. A JIT compiler complies Java bytecodes into instructions specified by the native instruction set of the hardware platform upon which execution is desired. While executing the compiled code is faster than execution in interpreter mode, the software compilation process itself is relatively time consuming. Thus, a large amount of memory is typically dedicated to storing the compiled code, so that the amount of time required to perform the compilation may be absorbed by performing the compilation once and allowing for the compiled code to be executed many times.

While the JIT compiler provides for speedier execution, the large amount of memory required to store the compiled code makes the JIT compiler unsuitable for certain types of machines. For example, set top boxes, personal digital assistants, and other hand-held computing devices generally have a limited amount of memory. Thus, dedicating a large amount of memory to store compiled Java code is not possible in these types of computing devices.

SUMMARY OF THE INVENTION

The problems outlined above are in large part solved by a system as described herein. The system

includes a CPU and a code translator. The CPU may execute instructions from a first instruction set, and the code translator may translate Java code sequences from Java bytecodes to code sequences having instructions defined in the first instruction set. The translated code sequences may be executed by the CPU.

The Java instruction set is a stack-based instruction set, and the first instruction set may be a register-based instruction set. Accordingly, the code translator may translate stack operand references to register operand references. More particularly, the code translator may include a stack transform storage which stores a mapping of register indexes to stack items forming the top of the stack. The code translator may assign register indexes to a particular instruction in the translated code sequence according to the mapping and any changes to the mapping performed by other instructions which are concurrently translated and prior to the particular instruction. Accordingly, instructions in the translated code sequence may use register operands and thus operand access may be efficient when the translated code sequence is executed on the CPU. While Java is used as an example of instructions which the code translator translates, the code translator may translate instructions from any instruction set to instructions executable by the CPU. Furthermore, the conversion of stack operand references to register operand references may be performed for any stack-based instruction set and register-based instruction set.

Broadly speaking, an apparatus is contemplated comprising a storage and a transform circuit coupled to the storage. The storage is configured to store a plurality of register indexes. The plurality of register indexes identify a plurality of registers storing a plurality of stack items comprising a top portion of a stack. Coupled to receive one or more instructions and corresponding stack change information, the transform circuit is configured to assign one or more of the plurality of register indexes to each of one or more source operands of the one or more instructions responsive to the stack change information.

Additionally, a method is contemplated. One or more instructions and corresponding stack change information are received. One or more register indexes are assigned to one or more source operands of the one or more instructions. The one or more register indexes are read from a storage and the one or more register indexes are indicative of registers storing one or more stack items forming a top portion of a stack.

BRIEF DESCRIPTION OF THE DRAWINGS

Other objects and advantages of the invention will become apparent upon reading the following detailed description and upon reference to the accompanying drawings in which:

Fig. 1 is a block diagram of a computing system.

Fig. 2 is a block diagram of an exemplary memory map for one embodiment of the computing system shown in Fig. 1.

Fig. 3 is a block diagram illustrating a storage model for a source instruction set and a target instruction set, according to one embodiment of the computing system shown in Fig. 1.

Fig. 4 is a flowchart illustrating operation of one embodiment of the computing system shown in Fig. 1 during invocation of a method.

Fig. 5 is a flowchart illustrating operation of one embodiment of the computing system shown in Fig. 1 in response to an interrupt from the code translator.

Fig. 6 is a block diagram illustrating one embodiment of translated and non-translated code streams.

Fig. 7 is a block diagram of one embodiment of the code translator shown in Fig. 1.

Fig. 8 is a block diagram of one embodiment of a translate unit shown in Fig. 7.

Fig. 9 is a block diagram of one embodiment of a stack to register transform unit shown in Fig. 8.

Fig. 10 is a block diagram of one embodiment of a fetch unit shown in Fig. 7.

Fig. 11 is a block diagram of one embodiment of a translate unit shown in Fig. 7.

Fig. 12 is a table illustrating assignment of source operands according to one embodiment of a stack to register transform unit shown in Fig. 9.

Fig. 13 is a table illustrating resulting stack transform according to one embodiment of a stack to register transform unit shown in Fig. 9 for a decode group of instructions.

Fig. 14 is a table illustrating resulting free list according to one embodiment of a stack to register transform unit shown in Fig. 9 for a decode group of instructions.

Fig. 15 is a flowchart illustrating operation of one embodiment of a decode unit shown in Fig 8.

Fig. 16 is an exemplary code sequence which may be produced by the decode unit shown in Fig. 8 according to the flowchart shown in Fig. 15.

Fig. 17 is a flowchart illustrating operation of one embodiment of a decode unit shown in Fig 8.

Fig. 18 is an exemplary code sequence which may be produced by the decode unit shown in Fig. 8 according to the flowchart shown in Fig. 17.

While the invention is susceptible to various modifications and alternative forms, specific embodiments thereof are shown by way of example in the drawings and will herein be described in detail. It should be understood, however, that the drawings and detailed description thereto are not intended to limit the invention to the particular form disclosed, but on the contrary, the intention is to cover all modifications, equivalents and alternatives falling within the spirit and scope of the present invention as defined by the appended claims.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Turning now to Fig. 1, a block diagram of one embodiment of a system 10 is shown. Other embodiments are possible and contemplated. The illustrated system 10 includes a central processing unit (CPU) 12, a memory controller 14, a memory 16, a Peripheral Component Interconnect (PCI) bridge 18, a PCI bus 20, a code translator 22, and an interrupt controller 24. CPU 12 is coupled to PCI bridge 18, memory controller 14, and interrupt controller 24. Memory controller 14 is further coupled to memory 16. PCI bridge 18 is further coupled to PCI bus 20. Code translator 22 is coupled to interrupt controller 24 and to PCI bus 20. In the illustrated embodiment, code translator 22 includes a source address register 26, a target address register 28, a control register 30, and a status register 32. In one embodiment, CPU 12, memory controller 14, and PCI bridge 18 may be integrated onto a single chip or into a package as illustrated by the dotted line surrounding these components in Fig. 1 (although other embodiments may provide these components separately).

Generally, CPU 12 is capable of executing instructions defined in a first instruction set (the native instruction set of system 10). The native instruction set may be any instruction set, e.g. the ARM instruction set, the PowerPC instruction set, the x86 instruction set, the Alpha instruction set, etc. Code translator 22 is provided for translating code sequences coded using a second instruction set, different from the native instruction set, to a code sequence coded using the native instruction set. Instruction sequences coded using the second instruction set are referred to as "non-native" code sequences, and code sequences coded using the first instruction set of CPU 12

are referred to as "native" code sequences.

When CPU 12 detects that a non-native code sequence is to be executed, CPU 12: (i) stores the source address of the non-native code sequence in source address register 26; (ii) stores the target address at which code translator 22 is to write the translated code sequence; and (iii) stores a command in control register 30 to activate code translator 22. In response to the command, code translator 22 reads the non-native code sequence from the source address, translates the non-native code sequence to a native code sequence, and stores the native code sequence at the target address. Code translator 22 provides a status for the translation in status register 32, and signals CPU 12 that the translation is complete. In the present embodiment, for example, code translator 22 may assert an interrupt signal to interrupt controller 24, which may subsequently interrupt CPU 12. CPU 12 may access interrupt controller 24 to determine the source of the interrupt (e.g. interrupt controller 24 may be coupled to PCI bus 20). In response to determining that the source of the interrupt is code translator 22, CPU 12 may read status register 32 to ensure that no errors occurred during the translation, and may then execute the native code sequence stored at the target address. It is noted that the source and target addresses are addresses identifying memory locations within the memory 16.

In one embodiment, code translator 22 is configured to translate Java code sequences to the native instruction set. Thus, Java bytecodes will be used as an example of a non-native instruction set below. However, the techniques described below may be used with any non-native instruction set. Additionally, the Java instruction set uses a stack-based programming and storage model, while the native instruction set may use a register-based programming and storage model. The techniques described below for converting between the Java instruction set and the register-based native instruction set are applicable to converting any other stack-based instruction set. As used herein, the term "stack-based programming and storage model" or "stack-based instruction set" refer to a model or instruction set in which operands for instructions are stored in a stack, generally in memory. Thus, execution of an instruction typically involves a memory reference for the operands (except for immediate operands). On the other hand, the terms "register-based programming and storage model" or "register-based instruction set" refer to a model or instruction set in which operands for instructions are stored in a set of registers defined by the architecture. Each register is identified via a register index, and the register indexes are coded into the instructions to specify the operands of the instructions. Operand fetch for instructions in a register-based instruction set are then generally reads of the registers, typically implemented within the CPU. Register-based instruction sets often use explicit load/store instructions to load operands from memory locations to registers for subsequent instructions to use as operands and to store results from registers to memory locations. Furthermore, the term "instruction set" as used herein refers to a group of instructions defined by a particular architecture. Each instruction in the instruction set may be assigned an opcode which differentiates the instruction from other instructions in the instruction set, and the operands and behavior of the instruction are defined by the instruction set. Thus, Java bytecodes are instructions within the instruction set specified by the Java language specification, and the term bytecode and instruction will be used interchangeably herein when discussing Java bytecodes. Similarly, ARM instructions are instructions specified in the ARM instruction set, PowerPC instructions are instructions specified in the PowerPC instruction set, etc.

Since code translator 22 may translate from a stack-based instruction set to a register-based instruction set, code translator 22 may include hardware for translating the stack references in the stack-based instruction set to

register indexes in the register-based instruction set. More particularly, a subset (or "pool") of the registers may be reserved to store stack operands. Code translator 22 may assign register indexes as values are pushed onto the stack, and may use those register indexes as source operands for instructions which reference the stack. After the values are popped from the stack, the corresponding registers may be free for use for another value pushed onto the stack. Thus, the register pool may store the topmost operands on the stack, and memory may be used for lower items (as will be described in more detail below). The register-based instruction set may be most efficient at accessing operands in registers (since loads and stores may be needed to read the values from memory), and thus keeping items at the top of the stack in registers may enhance performance.

As an alternative to reserving the pool of registers, code translator 22 may be configured to statically or dynamically allocate registers from the register set of CPU 12 into the register pool. Code translator 22 may generate native instructions to store the registers selected for the pool to a scratchpad memory area (preserving the values in the selected registers), and then these registers may be used to store stack items. In a static embodiment, the entire pool of registers may be allocated at the beginning of a translated code sequence. In a dynamic embodiment, registers may be allocated to the pool as additional registers are needed during the translation. At the end of the translated code sequence, code translator 22 may insert instructions to restore the values of these registers by reading the values from the scratchpad memory area (after storing the items to the operand stack).

In one embodiment, code translator 22 may translate instructions beginning at the source address and up to a basic block boundary in response to being activated by CPU 12. Generally, instructions within a basic block are not branch instructions (e.g. conditional or unconditional branches, call or return instructions, etc.). Once a basic block is entered, each instruction in the basic block is executed. The basic block boundary is formed by an branch instruction. Upon translating the branch instruction, code translator 22 may update status register 32 and assert the interrupt signal. Other embodiments may employ branch prediction and speculatively translate instructions past the basic block boundary based on the branch prediction. If the branch prediction is incorrect, the speculative translation may be discarded.

In another embodiment, code translator 22 may translate instructions through an unconditional branch, stopping translation when a conditional branch instruction or the end of the code sequence is encountered. The unconditional branch instruction may be deleted from the translated code sequence ("folded out") and the instructions at the target address of the unconditional branch instruction may be inserted in-line in the translated code (sequential to the instructions translated from the code preceding the unconditional branch instruction). Such an embodiment may further provide speculative translation beyond conditional branches, as mentioned above. Additionally, code translator 22 may limit the total number of instructions translated before stopping and signalling CPU 12. The total number may be the number of source instructions (e.g. non-native instructions) or the number of target instructions (e.g. native instructions). Alternatively, the number of bytes may be limited (and may be either the number of bytes of source instructions or the number or bytes of target instructions). The limit on the number of bytes/instructions may be programmable in a configuration register of code translator 22 (not shown). In one particular implementation, for example, a maximum size of 64 or 128 bytes of translated code may be programmably selected.

Because code translator 22 translates code in hardware, code translator 22 may be capable of producing native code sequences corresponding to Java code sequences more rapidly than a software JIT compiler.

Accordingly, system 10 need not dedicate a large amount of memory to store translated code sequences. Instead, a relatively small amount of memory may be used and additional sequences may be translated by code translator 22 as needed.

Generally, CPU 12 executes native code sequences and controls other portions of the system in response to the native code sequences. More particularly, CPU 12 may execute the JVM for system 10, including the interpreter mode to handle exception conditions detected by code translator 22. The JVM executed by CPU 12 may include all of the standard features of a JVM and may further include code to activate code translator 22 when a Java code sequence is to be executed, and to jump to the translated code after code translator 22 completes the translation. Code translator 22 may insert a return instruction to the JVM. CPU 12 may further execute the operating system code for system 10, as well as any native application code that may be included in system 10.

Memory controller 14 receives memory read and write operations from CPU 12 and PCI bridge 18 and performs these read and write operations to memory 16. It is noted that some of the read and write operations presented by PCI bridge 18 may be read and write operations generated by code translator 22 (e.g. read operations from the source address and subsequent addresses and write operations to the target address and subsequent addresses). Memory 16 may comprise any suitable type of memory, including SRAM, DRAM, SDRAM, RDRAM, or any other type of memory.

PCI bridge 18 facilitates communication between PCI bus 20 and memory controller 14 or CPU 12. More particularly, source address register 26, target address register 28, control register 30, and status register 32 may be memory-mapped registers. PCI bridge 18 may detect read or write operations to the addresses to which the registers are mapped, and transmit those operations on PCI bus 20 to code translator 22. As mentioned above, PCI bridge 18 may also detect read and write operations from code translator 22 to memory 16 on PCI bus 20 and may transmit those operations to memory controller 14.

Interrupt controller 24 generally receives interrupt signals from code translator 22 and other devices within system 10 (not shown), and prioritizes the interrupts received. If one or more interrupts have been signalled, interrupt controller 24 may assert the interrupt signal to CPU 12. CPU 12 may then access interrupt controller 24 to determine the source of the highest priority pending interrupt, and may service that interrupt.

It is noted that, while the PCI bus is used as an exemplary peripheral bus in the embodiment of Fig. 1, any other bus may be used. For example, the Universal Serial Bus (USB), IEEE 1394 bus, the Industry Standard Architecture (ISA) or Enhanced ISA (EISA) bus, the Personal Computer Memory Card International Association (PCMCIA) bus, etc. may be used. Still further, the Advanced RISC Machines (ARM) Advanced Microcontroller Bus Architecture (AMBA) bus, including the Advanced High-Performance (AHB) and/or Advanced System Bus (ASB) may be used, as may the Handspring Interconnect specified by Handspring, Inc. (Mountain View, CA). Still further, code translator 22 may be connected to memory 16 using a Unified Memory Architecture connection. In other alternatives, code translator 22 may be directly connected to CPU 12 or memory 16, or may be integrated into CPU 12, memory controller 14, or PCI bridge 18.

In other embodiments, interrupt controller 24 may be deleted and code translator 22 may assert an interrupt signal directly to CPU 12. Still further, other embodiments may employ semaphores in memory for communication between CPU 12 and code translator 22. Any technique for communicating code sequences to be translated and completion of the translation may be used.

Turning now to Fig. 2, a block diagram illustrating an exemplary contents of memory 16 is shown. More particularly, memory 16 in the example is storing a Java Virtual Machine (JVM) 40, a Java class 42 including a first Java method 44 and a second Java method 46, a translation cache table 48, and a scratch pad memory 50 including a translated method 52.

JVM 40 includes the native instructions to implement the Java Virtual Machine Specification, and further includes instructions to activate code translator 22 if Java bytecodes are to be executed. Flowcharts shown in Figs. 4-5 below may illustrate portions of JVM 40 used to interface with code translator 22.

Class 42 is an exemplary Java class including methods 44 and 46. Methods 44 and 46 are coded using Java bytecodes. On the other hand, translated method 52 in scratch pad memory 50 includes native instructions which, when executed, perform the same function in system 10 as method 44 would if executed in interpreter mode. Translated method 52 may comprise a portion of method 44, if code translator 22 has not yet translated all of method 44. Furthermore, if translated method 52 would exceed the size of scratch pad memory 50, the translation of a later portion of method 44 may overwrite the translation of an earlier portion of method 44 within scratch pad memory 50. Translated method 52 may comprise multiple code sequences, each terminated with a return instruction which returns to JVM 40. JVM 40 may then check the next address to be executed against translation cache table 48 to determine if the code sequence is already translated and residing in scratch pad memory 50. If the code sequence is already translated, JVM 40 calls the translated code sequence. Otherwise, JVM 40 activates code translator 22 to translate the code sequence.

Accordingly, if method 44 is to be translated by code translator 22, the source address stored into source address register 26 may be the entry point of method 44 within memory 16. The target address may be an address within scratch pad memory 52. Fig. 2 further illustrates that the translated method code is placed in different memory locations than the original method code. In this manner, the untranslated, non-native code sequence is available for interpreted execution in the event of an exception during the translation process.

Translation cache table 48 is used to store information related to translated methods. More particularly, translation cache table 48 may comprise a number of entries. Each entry may include a method reference identifying the method (or portion of the method if the method is translated in portions, e.g., because it includes conditional branches or is too long to translate as a whole). For example, the method reference may be the source address of the first instruction translated by the corresponding translated code sequence stored in scratch pad memory 52. The entry may further include a pointer to the translated code sequence (e.g. an address within scratch pad memory 52) and the size of the translated code sequence. Other information may be included as desired. It is noted that, in an embodiment in which a maximum size of the translated code is implemented, scratch pad memory 52 may be allocated in units of the maximum size and translation cache table 48 may include an entry for each unit within scratch pad memory 52.

It is noted that one or more memory locations within memory 16 may correspond to memory-mapped registers (e.g. registers 26-32, as well as any additional configuration/control registers which may be implemented according to various embodiments of code translator 22).

Turning now to Fig. 3, a diagram illustrating a stack-based programming and storage model (e.g. the Java programming and storage model) and a register-based programming and storage model (e.g. the programming and storage model of CPU 12) when executing translated Java code sequences is shown. In the stack-based

programming and storage model, a stack 60 is stored in memory 16. A top of stack (TOS) pointer is maintained by the JVM 40 which identifies the memory location storing the stack item which is at the top of the stack. The TOS pointer may be more succinctly referred o as the stack pointer, and is stored in a register 62. Register 62 may be one of the registers in the register set of CPU 12, for a JVM implemented as native code operating on CPU 12. As items are pushed onto the stack, the items are stored into memory locations contiguous to the memory location indicated by the stack pointer, and the stack pointer is updated to indicate the new top of stack. As items are popped from the stack, the stack pointer is updated to indicate the new top of stack (e.g. if item S(0) is popped in Fig. 3, the stack pointer is updated to indicate S(1)).

The stack 60 is represented in the register-based based programming and storage model (after the corresponding code is translated by code translator 22) by a register pool 64, a stack transform 66, a stack 68, and a memory top of stack (MTOS) pointer 70. As Fig. 3 illustrates, a portion of the top of stack 60 are stored in registers within register pool 64 (which is a subset of the registers included in CPU 12). Accordingly, access to the operands at the top of the stack may be efficient in a register-based programming and storage model, since these operands are stored in registers. Operands further down the stack are stored in stack 68, with the MTOS pointer 70 indicating the top of the stack 68 (e.g. item S(4) in this example, where items S(0) through S(3) are stored in registers). It is noted that MTOS pointer 70 may be stored in another register within the register set, outside of the register pool 64. More particularly, MTOS pointer 70 may be stored in register 62, and may be the stack pointer prior to entry into the translated code, in one embodiment. In such an embodiment, updates to the stack pointer register stored in register 62 may be deferred until the translated code sequence is terminated. As items are pushed onto the stack, registers from register pool 64 are allocated to store the items. As items are popped from the stack, the registers storing those items become free for allocation during a subsequent push.

Code translator 22 manages the register pool 64 and assigns register indexes for the operands of instructions in the translated Java code sequences based on which registers are storing the top of stack operands. Code translator 22 maintains stack transform 66, which maps the stack locations of stack 60 to the register indexes of the registers in register pool 64 assigned to those stack locations. For example, register 72 in Fig. 3 is storing the top stack item S(0). Stack transform 66 thus provides the register index identifying register 72 for instructions needing the top of stack value as an operand.

Code translator 22 may also handle the overflow and underflow of register pool 64. If a push is detected in the code sequence and all the registers in register pool 64 are storing stack items (overflow), one or more registers in the register pool may be freed by pushing the values in those registers onto stack 68. More particularly, the registers storing the stack items farthest from the top of the stack may be freed in this fashion. Code translator 22 may automatically generate the store instructions (to be executed by CPU 12) to push the values onto stack 68 and free the registers (storing these instructions at the target address along with the instructions representing the translated Java code sequence), or may generate an interrupt to CPU 12 and have an interrupt service routine provide the instructions to push the values to memory. Alternatively, code translator 22 may monitor the number of free registers available and, when the number is less than a threshold value, free some of the registers by pushing their contents to stack 68. Similarly, if the registers in register pool 64 are storing no stack items (underflow), code translator 22 may generate instructions to load the values from the top of stack 68 into the registers (or may use an interrupt to CPU 12 similar to the above description).

Turning next to Fig. 4, a flowchart illustrating certain operations of one embodiment of JVM 40 when invoking a method is shown. Other embodiments are possible and contemplated. The steps shown in Fig. 4 are illustrated in a particular order for ease of understanding. However, any suitable order may be used.

When invoking a method, JVM 40 determines if the method has previously been translated by code translator 22 and still remains within scratch pad memory 50 (decision block 100). More particularly, JVM 40 scans translation cache table 48 to determine if the method is recorded in the table. If the method has been translated, JVM 40 branches to the pointer from translation cache table 48 and executes the translated code (step 108). On the other hand, if the method has not been translated, JVM 40 activates code translator 22. More particularly, JVM 40 stores the source address of the method in source address register 26 (step 102), the target address at which the translated method is to be written within scratch pad memory 50 into target address register 28 (step 104), and the command into control register 30 which initiates code translation in code translator 22 (the "go" command) (step 106). JVM 40 then waits for a signal from code translator 22 that the translation is complete (e.g. an interrupt is signalled). JVM 40 may perform other activities while waiting for the signal, if desired.

Turning next to Fig. 5, a flowchart illustrating certain operations of one embodiment of JVM 40 when an interrupt from code translator 22 is received is shown. Other embodiments are possible and contemplated. The steps shown in Fig. 5 are illustrated in a particular order for ease of understanding. However, any suitable order may be used. Prior to performing the actions illustrated in Fig. 5, JVM 40 may determine that the interrupt is from code translator 22, if CPU 12 may receive interrupts from multiple sources within system 10.

JVM 40 reads status register 32 to determine if the translation completed successfully (step 120). There may be a variety of reasons why the translation could not be completed successfully. For example, certain embodiments of code translator 22 may signal an interrupt with unsuccessful translation if an underflow or overflow of register pool 64 occurs. Additionally, certain embodiments of code translator 22 may signal an interrupt if the translated code sequence exceeds the size of scratch pad memory 50 or the maximum size of a translated code sequence. Other embodiments may signal an interrupt to handle certain Java instruction encodings (bytecodes) which may be too difficult to translate in hardware. These bytecodes may be executed in the interpreter, and then code translator 22 may be reactivated to continue translating the subsequent bytecodes. Code translator 22 may be configured to generate instructions in the translated code sequence which store the stack items which are in registers of CPU 12 to the operand stack in memory ("spill the registers to the operand stack") and update the stack pointer maintained by the JVM to reflect the current stack state. Additionally, the code translator may update another register of CPU 12 which stores the program counter (PC) of the Java code sequence for the JVM, to reflect the instructions translated by code translator 22. In this manner, the stack pointer and PC may reflect the operation of the Java instructions translated by code translator 22.

If status register 32 indicates unsuccessful translation (decision block 122), JVM 40 may execute the method (or the portion of the method which is unsuccessfully translated, e.g. beginning at the source address stored in source address register 26 prior to the exception) in interpreter mode to handle the exception condition (step 124). Fig. 6 below illustrates the handling of exceptions detected by code translator 22.

If status register 32 indicates no exception, JVM 40 may update the translation cache table 48 with information indicating the source method, a pointer to the translated code, the size, etc (step 118). JVM 40 may manage translation cache table 48 in any suitable fashion. For example, entries in translation cache table 48 may be

managed in a first-in, first-out (FIFO) fashion, reusing the oldest entry after each entry has been used. Alternatively, entries may be managed in a least recently used (LRU) fashion.

JVM 40 may determine if there is additional code to translate (decision block 126). If so, the source address, target address, and command value may be stored in source address register 26, target address register 28, and control register 30, respectively (steps 102-106). Additionally, JVM 40 may branch to the translated code and execute the code (step 128).

It is noted that the flowchart shown in Fig. 5 may represent speculation on the part of JVM 40 that the translated code sequence executes properly. Furthermore, JVM 40 may speculate on the direction of a conditional control-flow instruction to determine the next code sequence to translate. Other embodiments may execute the translated code sequence first, then activate code translator 22 to translate the next code sequence to be executed.

Fig. 6 is a block diagram illustrating the handling of exceptions detected by code translator 22. Illustrated in Fig. 6 is a Java code stream 130 which includes a call to Java method 132. A corresponding translated Java code stream 134 generated by code translator 22 in response to Java code stream 130 and a translated method 136 generated by code translator 22 in response to Java method 132 are also shown. The call, from Java code stream 130, to Java method 132 is illustrated by solid arrow 138. A corresponding call from translated code stream 134 to translated method 136 is illustrated by solid arrow 140. Dotted arrow 142 illustrates the detection, by code translator 22 during translation of method 132 to method 136, of an exception. The exception causes JVM 40 to execute Java method 132 in interpreter mode (dotted arrow 144). It is noted that, if a portion of method 132 has been translated and executed successfully, JVM 40 may execute, in interpreter mode, the portion of the method for which an exception was detected during translation (rather than executing the entire method in interpreter mode).

Turning next to Fig. 7, a block diagram of one embodiment of code translator 22 is shown. Other embodiments are possible and contemplated. In the embodiment of Fig. 7, code translator 22 includes a PCI interface unit 150, a fetch unit 152, a translate unit 154, a write unit 156, source address register 26 (within fetch unit 152), target address register 28 (within write unit 156), control register 30, and status register 32. PCI interface unit 150 is coupled to fetch unit 152, translate unit 154, write unit 156, control register 30, status register 32, source address register 26, target address register 28, PCI bus 20, and an interrupt line 158. Fetch unit 152 is further coupled to translate unit 154 and control register 30. Translate unit 154 is further coupled to status register 32 and write unit 156.

Generally speaking, in response to a "go" command written into control register 30, fetch unit 152 is configured to begin fetching source instructions. Fetch unit 152 may initiate fetching from the source address, and may receive fetch addresses from translate unit 154. Additionally, fetch unit 152 may be configured to prefetch addresses ahead of translate unit 154 requests, if desired. Translate unit 154 may receive the source instructions and may predecode the source instructions to determine stack change information corresponding to each instruction. Additionally, translate unit 154 may decode the source instructions into target instructions. Translate unit 154 translates the stack operand references of the source instructions into register indexes for the target instructions. The target instructions, with register operand assignments, are then provided to write unit 156. Write unit 156 provides the target instructions to PCI interface unit 150 along with a target address (initially the address in target address register 156 and subsequently incremented as instructions are stored out). PCI interface unit 150 writes the translated instructions to memory 16 via PCI bus 20. Once the translation of the code sequence is

stopped (e.g. an exception condition or basic block boundary is detected, the maximum translation size is reached, etc.), translate unit 154 updates status register 32 with the status of the translation. Additionally, translate unit 154 may generate a return instruction to the JVM. Responsive to the status being updated (and subsequent to completing the write commands from write unit 156), PCI interface unit 150 may assert the interrupt signal on interrupt line 158 to interrupt CPU 12 for execution of the translated code sequence.

For the description of portions of one embodiment of code translator 22 provided with respect to Figs. 7-18, the terms "source instructions" and "target instructions" will be used to refer to instructions fetched by code translator 22 and generated by code translator 22, respectively. For a system embodiment similar to system 10 shown in Fig. 1, source instructions may be non-native instructions (e.g. Java bytecodes), and target instructions may be native instructions for CPU 12.

As used herein, the term "stack change information" refers to information indicative of a modification of an operand stack by a corresponding source instruction. The stack change information may take any suitable form. In one embodiment, the stack change information may include a number of pushes performed by the source instruction, a number of pops performed by the source instruction, and a stack pointer modification by the source instruction (e.g. the difference between the number of pushes and the number of pops, or vice versa). Other embodiments may include alternative encodings of the stack change information, including any subset of the above information.

As mentioned above with respect to Fig. 1, while PCI interface unit 150 is shown in the present embodiment (e.g. with respect to Figs. 7-18), other embodiments may use any suitable external interface. For example, the Universal Serial Bus (USB), IEEE 1394 bus, the Industry Standard Architecture (ISA) or Enhanced ISA (EISA) bus, the Personal Computer Memory Card International Association (PCMCIA) bus, etc. may be used. Still further, the Advanced RISC Machines (ARM) Advanced Microcontroller Bus Architecture (AMBA) bus, including the Advanced High-Performance (AHB) and/or Advanced System Bus (ASB) may be used, as may the Handspring Interconnect specified by Handspring, Inc. (Mountain View, CA).

Turning next to Fig. 8, a block diagram of one embodiment of translate unit 154 is shown. Other embodiments are possible and contemplated. In the embodiment of Fig. 8, translate unit 154 includes a predecode unit 160, a decode unit 162, and a stack to register transform unit 164. Stack to register transform unit 164 is coupled to decode unit 162, write unit 156, and fetch unit 152. Decode unit 162 is further coupled to predecode unit 160, which is further coupled to PCI interface unit 150.

Generally, fetch unit 152 is configured to begin fetching source instructions from the source address stored in source address register 26 responsive to detecting the "go" command in control register 30. As the code is translated by translate unit 154, stack to register transform unit 164 may generate fetch addresses for fetch unit 152. Fetch unit 152 may continue to generate addresses until directed to stop fetching by translate unit 154 (via a stop fetch signal illustrated in Fig. 8).

Predecode unit 160 is coupled to receive the source instructions from PCI interface 150, and predecodes each source instruction to determine stack change information. Predecode unit 160 supplies the source instructions to decode unit 162 along with the stack change information. Decode unit 162 generates target instructions for each source instruction, and provides the target instructions and the stack change information to stack to register transform unit 164. The stack change information may then be used to assign register operands to the target

instructions in stack to register transform unit 164.

It is noted that more than one target instruction may be generated for various source instructions. The stack change information corresponding to each target instruction may be derived from the stack change information provided by predecode unit 160. For example, in one embodiment, each target instruction may perform at most one push or one pop. In such embodiments, sufficient target instructions to perform each push and pop as specified by the source instruction may be generated.

Predecode unit 160 may be implemented in any suitable fashion. For example, predecode unit 160 may comprise a programmable logic array (PLA) structure, combinatorial logic, or a lookup table (either a read-only memory (ROM) lookup table, or a random access memory (RAM) lookup table). In lookup table form, each byte code could be assigned an entry in the table, with the number of pushes, the number of pops, and the stack pointer modification stored in the entry. Additionally, Java bytecodes may include a wide prefix which indicates that the operands each occupy two stack entries. Accordingly, if the wide prefix is included, the values from the lookup table may be left-shifted by one bit to double the numbers. The left-shift may be performed in the senseamps at the output of the table, or via muxes outside the table, as desired. Table 1 below illustrates exemplary values stored in the lookup table for one embodiment of predecode unit 160 which may produce a number of pushes, number of pops, and a stack pointer modification for Java bytecodes. As mentioned above, certain Java byte codes may not be translated by code translator 22. Those instructions are indicated by "NT" in the number of pushes, number of pops, and stack pointer modification columns of table 1. Which instructions are not translated may be varied from embodiment to embodiment, including embodiments which translate all instructions. In another alternative, predefined code sequences ("macros") may be stored in memory 16 for one or more Java byte codes (e.g. byte codes which are complex to translate in hardware but also frequently used). When code translator 22 encounters a byte code for which a macro is provided, code translator 22 may generate a branch to the macro. The macro may return to the next instruction in the translated code sequence after completing execution.

Table 1: Exemplary Table of Predecode Information

Bytecode	# of Pushes	# of Pops	Stack Pointer Modification
aconst_null	1	0	+1
iconst_m1	1	0	+1
iconst_0	1	0	+1
iconst_1	1	0	+1
iconst_2	1	0	+1
iconst_3	1	0	+1
iconst_4	1	0	+1
iconst_5	1	0	+1
lconst_0	1	0	+1
lconst_1	1	0	+1
fconst_0	1	0	+1
fconst_1	1	0	+1

Table 1: Exemplary Table of Predecode Information (CONTINUED)

fconst_2	1	0	+1
dconst_0	1	0	+1
dconst_1	1	0	+1
bipush	1	0	+1
sipush	1	0	+1
nop	0	0	0
pop	0	1	-1
pop2	0	2	-2
dup	2	1	+1
dup_x1	3	2	+1
dup_x2	4	3	+1
dup2	4	2	+2
dup2_x1	5	3	+2
dup2_x2	6	4	+2
swap	2	2	0
iload	1	0	+1
iload_w	2	0	+2
lload	2	0	+2
lload_w	4	0	+4
fload	1	0	+1
fload_w	2	0	+2
dload	2	0	+2
dload_w	4	0	+4
aload	1	0	+1
aload_w	2	0	+2
iload_0	1	0	+1
iload_1	1	0	+1
iload_2	1	0	+1
iload_3	1	0	+1
lload_0	2	0	+2
lload_1	2	0	+2
lload_2	2	0	+2
lload_3	2	0	+2
fload_0	1	0	+1
fload_1	1	0	+1
fload_2	1	0	+1
fload_3	1	0	+1

Table 1: Exemplary Table of Predecode Information (CONTINUED)

dload_0	2	0	+2
dload_1	2	0	+2
dload_2	2	0	+2
dload_3	2	0	+2
aload_0	1	0	+1
aload_1	1	0	+1
aload_2	1	0	+1
aload_3	1	0	+1
istore	0	1	-1
istore_w	0	2	-2
lstore	0	1	-1
lstore_w	0	2	-2
fstore	0	1	-1
fstore_w	0	2	-2
lstore	0	2	-2
lstore_w	0	4	-4
astore	0	1	-1
astore_w	0	2	-2
istore_0	0	1	-1
istore_1	0	1	-1
istore_2	0	1	-1
istore_3	0	1	-1
lstore_0	0	2	-2
lstore_1	0	2	-2
lstore_2	0	2	-2
lstore_3	0	2	-2
fstore_0	0	1	-1
fstore_1	0	1	-1
fstore_2	0	1	-1
fstore_3	0	1	-1
dstore_0	0	2	-2
dstore_1	0	2	-2
dstore_2	0	2	-2
dstore_3	0	2	-2
astore_0	0	1	-1
astore_1	0	1	-1
astore_2	0	1	-1

Table 1: Exemplary Table of Predecode Information (CONTINUED)

astore_3	0	1	-1
iaload	1	0	+1
laload	2	0	+2
faload	1	0	+1
daload	2	0	+2
aaload	1	0	+1
baload	1	0	+1
caload	1	0	+1
saload	1	0	+1
iastore	0	3	-3
lastore	0	4	-4
fastore	0	3	-3
dastore	0	4	-4
aastore	0	3	-3
bastore	0	3	-3
castore	0	3	-3
sastore	0	3	-3
ldc	1	0	+1
ldc_w	1	0	+1
ldc2_w	2	0	+2
iadd	1	2	-1
ladd	2	4	-2
fadd	1	2	-1
dadd	2	4	-2
isub	1	2	-1
lsub	2	4	-2
fsub	1	2	-1
dsub	2	4	-2
imul	1	2	-1
lmul	2	4	-2
fmul	1	2	-1
dmul	2	4	-2
idiv	1	2	-1
ldiv	2	4	-2
fdiv	1	2	-1
ddiv	2	4	-2
irem	1	2	-1

Table 1: Exemplary Table of Predecode Information (CONTINUED)

lrem	2	4	-2
frem	1	2	-1
drem	2	4	-2
ineg	1	1	0
lneg	2	2	0
fneg	1	1	0
dneg	2	2	0
iinc	0	0	0
ishl	1	2	-1
lshl	2	3	-1
ishr	1	2	-1
lshr	2	3	-1
iushr	1	2	-1
lushr	2	3	-1
iand	1	2	-1
land	2	4	-2
ior	1	2	-1
lor	2	4	-2
ixor	1	2	-1
lxor	2	4	-2
i2l	2	1	+1
i2f	1	1	0
i2d	2	1	+1
l2i	1	2	-1
l2f	1	2	-1
l2d	2	2	0
f2i	1	1	0
f2l	2	1	+1
f2d	2	1	+1
d2i	1	2	-1
d2l	2	2	0
d2f	1	2	-1
i2b	1	1	0
i2c	1	1	0
i2s	1	1	0
lcmp	1	4	-3
fcmpl	1	2	-1

Table 1: Exemplary Table of Predecode Information (CONTINUED)

fcmpg	1	2	-1
dcmpl	1	4	-3
dcmpg	1	4	-3
ifeq	0	1	-1
ifne	0	1	-1
iflt	0	1	-1
ifge	0	1	-1
ifgt	0	1	-1
ifle	0	1	-1
if_icmpeq	0	2	-2
if_icmpne	0	2	-2
if_icmplt	0	2	-2
if_icmpge	0	2	-2
if_icmpgt	0	2	-2
if_icmple	0	2	-2
if_acmpeq	0	2	-2
if_acmpne	0	2	-2
goto	0	0	0
jsr	1	0	+1
ret	0	0	0
ret_w	0	0	0
ifnull	0	1	-1
ifnonnull	0	1	-1
goto_w	NT	NT	NT
jsr_w	NT	NT	NT
tableswitch	NT	NT	NT
lookupswitch	NT	NT	NT
ireturn	NT	NT	NT
lreturn	NT	NT	NT
freturn	NT	NT	NT
dreturn	NT	NT	NT
areturn	NT	NT	NT
return	NT	NT	NT
getstatic	NT	NT	NT
putstatic	NT	NT	NT
getfield	NT	NT	NT
putfield	NT	NT	NT

Table 1: Exemplary Table of Predecode Information (CONTINUED)			
invokevirtual	NT	NT	NT
invokespecial	NT	NT	NT
new	NT	NT	NT
newarray	NT	NT	NT
anewarray	NT	NT	NT
arraylength	0	0	0
athrow	NT	NT	NT
checkcast	NT	NT	NT
instanceof	NT	NT	NT
monitorenter	NT	NT	NT
monitorexit	NT	NT	NT
multinewarray	NT	NT	NT
breakpoint	NT	NT	NT
ldc_quick	NT	NT	NT
ldc_w_quick	NT	NT	NT
ldc2_w_quick	NT	NT	NT
getfield_quick	NT	NT	NT
putfield_quick	NT	NT	NT
getfield2_quick	NT	NT	NT
pufield2_quick	NT	NT	NT
getstatic_quick	NT	NT	NT
putstatic_quick	NT	NT	NT
getstatic2_quick	NT	NT	NT
putstatic2_quick	NT	NT	NT
invokevirtual_quick	NT	NT	NT
invokenonvirtual_quick	NT	NT	NT
invokesuper_quick	NT	NT	NT
invokestatic_quick	NT	NT	NT
invokeinterface_quick	NT	NT	NT
invokevirtualobject_quick	NT	NT	NT
new_quick	NT	NT	NT
anewarray_quick	NT	NT	NT
multianewarray_quick	NT	NT	NT
checkcast_quick	NT	NT	NT
instanceof_quick	NT	NT	NT
invokevirtual_quick_w	NT	NT	NT
getfield_quick_w	NT	NT	NT

Table 1: Exemplary Table of Predecode Information (CONTINUED)			
putfield_quick_w	NT	NT	NT
impdep1	NT	NT	NT
impdep2	NT	NT	NT

Decode unit 162 may be implemented in any suitable fashion, similar to predecode unit 160. For example, decode unit 162 may comprise a programmable logic array (PLA) structure, combinatorial logic, or a lookup table (either a read-only memory (ROM) lookup table, or a random access memory (RAM) lookup table). In lookup table form, each byte code could be assigned an entry in the table, with the corresponding set of target instructions stored in the entry.

In the illustrated embodiment, decode unit 162 is coupled to a virtual stack pointer (VSP) register 166, a virtual program counter (VPC) register 167, and a spill count register 168. Decode unit 162 may use these registers to assist in generating target instructions for the translated code sequence. More particularly, decode unit 162 may use the VSP register 166 and the VPC register 167 to defer updates to the registers which JVM 40 uses to store the stack pointer to the operand stack and the PC of the Java code sequence, respectively. Rather than generate target instructions which update the PC and stack pointer registers as part of the target instructions corresponding to each source instruction, code translator 22 may record the cumulative updates to these registers for the source instructions which have been processed by decode unit 162. Decode unit 162 is coupled to receive the stop fetch signal from stack to register transform unit 164, and in response to the stop fetch signal, decode unit 162 may generate target instructions which add the cumulative update to the stack pointer (from VSP register 166) to the stack pointer register and add the cumulative update to the PC (from VPC register 167) to the PC register. Thus, these instructions may update the stack pointer register and PC register to reflect the effects of the source instructions which have been translated to target instructions in the translated code sequence.

Decode unit 162 may use the stack pointer modification provided by predecode unit 160 for each source instruction to generate the cumulative stack pointer modification for the source instructions processed in a particular clock cycle. Decode unit 162 may add the stack pointer modifications provided by predecode unit 162 to the current value in VSP register 166 to generate the updated value for VSP register 166, and may store that value back into VSP register 166. On the other hand, decode unit 162 may determine the PC updates by decoding the source instructions. Decode unit 162 may determine the length of each instruction, and may add the lengths of one or more source instructions processed during the clock cycle to the current value stored in VPC register 167 to generate the updated value for VPC register 167. The updated value may be stored back into VPC register 167. Accordingly, VSP register 166 may indicate the cumulative effect on the stack pointer of source instructions processed by decode unit 162 since being activated by CPU 12. Similarly, VPC register 167 may reflect the cumulative effect on the PC of source instructions processed by decode unit 162 since being activated by CPU 12. Subsequent to generating the target instructions to update the PC and stack pointer registers, decode unit 160 may clear VSP register 166 and VPC register 167 to prepare for translating the next code sequence on the next activation. Alternatively, registers 166 and 167 may be cleared when code translator 22 is activated to translate another code sequence.

Decode unit 162 may use spill count register 168 to support dynamic allocation of registers to the register

pool for storing stack items. Upon activation to translate a code sequence, decode unit 162 may allocate one or more registers for use to store stack items. Stack to register transform unit 164 may allocate the actual register indexes (e.g., counting backward from the highest register index), but decode unit 162 may control the number of registers allocated. Decode unit 162 may generate instructions to store the current contents of the registers to scratchpad memory 50, and may keep a count of registers thus freed in spill count register 168 (and may signal stack to register transform unit 164 with an indication that a register has been freed). Decode unit 162 may monitor the cumulative stack modification stored in VSP register 166. If the cumulative modification indicates that the number of items pushed onto the stack exceeds the number of items popped by a number close to or equal to the number of registers allocated (as indicate by spill count register 168), decode unit 162 may generate additional instructions to allocate registers into the register pool and may update spill count register 168 accordingly. When decode unit 162 receives an asserted stop fetch signal, decode unit 162 may generate instructions to restore the allocated registers, using the spill count to indicate how many instructions to generate. Additional details regarding dynamic allocation of registers into the register pool are provided further below. It is noted that, for embodiments which statically allocate registers into the register pool at the beginning of a translated code sequence or embodiments in which the register pool is reserved by the JVM for used by code translator 22, spill count register 168 may not be needed and may be eliminated.

Turning next to Fig. 9, a block diagram of one embodiment of stack to register transform unit 164 is shown. Other embodiments are possible and contemplated. In the embodiment of Fig. 9, stack to register transform unit 164 includes a translate control circuit 170, a free list 172, a stack transform storage 174, a transform circuit 176, and a top of stack pointer 178. Transform circuit 176 may comprise a register assign circuit 180 and a final stack transform circuit 182. Translate control circuit 170 is coupled to free list 172, stack transform storage 174, and fetch unit 152. Free list 172 is further coupled to transform circuit 176. Stack transform storage 174 is further coupled to transform circuit 176 and to top of stack pointer 178. Transform circuit 176 is further coupled to write unit 156.

Generally, stack transform storage 174 stores the mapping of register indexes to stack items which represents the state of the stack corresponding to instructions which have previously been processed by stack to register transform unit 164. Free list 172 stores the register indexes from register pool 64 which are free for assignment to newly pushed stack items. Each clock cycle, stack transform storage 174 and free list 172 provide register indexes to transform circuit 176 for assignment as operands of instructions, and free list 172 and stack transform storage 174 are updated to reflect the effects on the stack of the instructions processed during that clock cycle.

In one embodiment, stack transform storage 174 may comprise a register file or RAM for storing register indexes. The register file may be operated as a wrap around buffer, with the current top of stack indicated by top of stack pointer 178. In another embodiment, stack transform storage 174 may be configured to store multiple indexes in each entry. The entry including the index indicated as the top of stack and the subsequent entry may be read each clock cycle, and the indexes which comprise the top of stack may be provided from the indexes stored in the two entries. In one embodiment, free list 172 may be a FIFO storing the free register indexes and presenting the register indexes at the head of the list to transform circuit 176.

Register assign circuit 180 assigns register indexes for the source and destination operands of a particular

target instructions based on the stack change information of the instructions preceding that particular target instruction within the same decode group (concurrently received from decode unit 162) and based on the free list and stack transform information provided by free list 172 and stack transform storage 174, respectively. For example, the second instruction (in program order) of a decode group is assigned register indexes based on the stack transform and free list information, modified by the effects on the stack of the first instruction (as indicated by the stack change information corresponding to the first instruction). Similarly, the third instruction of the decode group is assigned register indexes based on the stack transform and free list information, modified by the effects on the stack of the first and second instructions (as indicated by the stack change information corresponding to the first and second instructions). Register assign circuit 180 provides the register indexes and target instructions to write unit 156.

More particularly, register assign circuit 180 is configured, for each instruction, to assign source operand register indexes of the registers which are the top of the stack and the next to the top of the stack (as modified by the preceding instructions within the decode group). The destination operand register index for each instruction is the head of the free list (as updated to delete register indexes consumed for destination operands of preceding instructions within the decode group). It is noted that a particular instruction may have a destination operand if that instruction pushes a value onto the operand stack as defined by the source instruction set.

As an example, Fig. 12 may be a truth table illustrating operation of one embodiment of register assign circuit 180 for assigning source operands to the third target instruction (instruction 2) in a decode group. In the embodiment shown, each target instruction may cause one push, one pop, or no stack change. Each row of the table illustrates the number of pushes and pops caused by each of instructions 0 and 1, and the resulting source operand assignment for instruction 2 (Src0 and Src1). The source operand assignments are listed in terms of stack transform information prior to the effects of instructions within the decode group (S[number]) or free list information prior to the effects of instructions within the decode group (F[number]). More particularly, S[0] may be the register index corresponding to the stack item at the top of the stack (as indicated by the stack transform information), S[1] may be the register index corresponding to the stack item second to the top of the stack (as indicated by the stack transform information), S[2] may be the register index corresponding to the stack item third to the top of the stack (as indicated by the stack transform information), etc. Similarly, F[0] may be the register index at the head of the free list, F[1] may be the register index second to the head of the free list, etc.

It is noted that the first three rows of Fig. 12 (which show instruction 1 as causing zero pushes and zero pops) may also illustrate source operand assignment for instruction 1, based on the pushes and pops of instruction 0 and the stack transform and free list information prior to the effects of instruction 1. While the table illustrated in Fig. 12 illustrates source operand assignment for the third instruction, other embodiments may include more than three instructions in a decode group. Thus, the table shown in Fig. 12 is merely exemplary.

Final stack transform circuit 182 computes the updated stack transform and free list information for update into stack transform storage 174 and free list 172, respectively. For example, final stack transform circuit 182 may indicate to free list 172 how many register indexes were consumed from the head of the free list, and may provide register indexes to be added to the end of the free list (registers which stored stack items which were popped). Final stack transform circuit 174 may further provide a new top of stack pointer for top of stack pointer 178 and may provide an updated list of register indexes to stack transform storage 174.

As an example, Figs. 13 and 14 may be truth tables which illustrate the resulting stack transform (Fig. 13) and the resulting free list (Fig. 14) from a decode group of three instructions (instruction 0, instruction 1, and instruction 2) based on the stack transform and free list prior to the effects of the three instructions according to one embodiment of final stack transform circuit 182. Similar to the table shown in Fig. 12, each target instruction may cause one push, one pop, or no stack change. Each row of the table illustrates the number of pushes and pops caused by each of instructions 0, 1, and 2, and the resulting stack transform and free list for that set of pushes and pops. The resulting stack transform and resulting free list are listed in terms of stack transform information prior to the effects of instructions within the decode group (S[number]) or free list information prior to the effects of instructions within the decode group (F[number]). More particularly, S[0] may be the register index corresponding to the stack item at the top of the stack (as indicated by the stack transform information), S[1] may be the register index corresponding to the stack item second to the top of the stack (as indicated by the stack transform information), S[2] may be the register index corresponding to the stack item third to the top of the stack (as indicated by the stack transform information), etc. Similarly, F[0] may be the register index at the head of the free list. F[1] may be the register index second to the head of the free list, etc.

The resulting stack transform illustrated in the table of Fig. 13 shows the four top items of the stack transform, with the top item on the left of the list and other items increasingly away from the top as the list progresses to the right. Since each instruction may include at most one push or one pop, the remaining elements of the stack transform below those shown will be the elements in increasing order from the last element shown (e.g. in the first row, the fifth element in the stack transform is S[4] and the sixth element is S[5], etc., while in the second row, the fifth element in the stack transform is S[5] and the sixth element is S[6], etc.).

The resulting free list illustrated in the table of Fig. 14 shows the list (with the head of the list on the left and increasing in order to the tail of the list on the right). Items indicated by ellipses are F[4], F[5], and F[6], in that order.

It is noted that the first 9 rows of the tables in Figs 13 and 14 illustrate the resulting stack transform and resulting free list for a decode group having two instructions (respectively), and the first 3 rows of the tables illustrate the resulting stack transform and resulting free list for a decode group having one instruction (respectively). Additionally, the tables may be expanded to handle decode groups having four or more instructions.

Translate control circuit 170 is coupled to receive the target instructions from decode unit 162, and is further coupled to receive a free list empty signal from free list 172 and a stack empty signal from stack transform storage 174. If either the free list is empty or the stack is empty, translate control circuit 170 may terminate translation of the code and store an exception encoding in status register 32. CPU 12 may service the interrupt by pushing register values to memory (free list empty) or loading values from memory (stack empty) to allow translation to continue. Alternatively, translate control circuit may generate these instructions automatically rather than causing an exception. Additionally, translate control circuit 170 may be configured to detect other exceptions (e.g. instructions which are not translated by translate control circuit 170) and to terminate translation and store an exception encoding in status register 32 for those exceptions. A different exception encoding may be provided for each type of exception.

Additionally, translate control circuit 170 may determine, from an examination of the target instructions, that the translation is complete. For example, in one embodiment, code translator 22 translates up to a conditional

branch or a maximum number of bytes in the source or target sequence. If translate control circuit 170 determines that the translation is complete, translate control circuit 170 terminates translation and stores a non-exception status encoding into status register 32.

If translate control circuit 170 terminates translation, it also asserts a stop fetch signal to fetch unit 152 to terminate additional fetching of source instructions. On the other hand, if translation is to continue, translate control circuit 170 may determine the next fetch address by examining the instructions currently being operated upon by translate unit 154. The next fetch address may generally be the address sequential to the last instruction in the current decode group, or may be the target address of a branch instruction if a branch instruction is encountered. The target address may be generated by translate control circuit 170 by adding the source address of the opcode of the branch instruction (which may be provided along with the decode group) to the displacement field of the branch instruction (one or more bytes following the branch instruction opcode).

After assertion of the stop fetch signal, stack to register transform unit 164 may continue to perform register assignment for target instructions provided by decode unit 162 (e.g. stack and PC register adjust instructions and restore instructions).

Turning next to Figs. 10 and 11, a second embodiment of a portion of fetch unit 152 (Fig. 10) and translate unit 154 (Fig. 11) is shown. The embodiment illustrated in Figs. 11 and 12 may provide for speculative translation past a conditional branch instruction. More particularly, the embodiment of translate unit 154 shown in Fig. 11 includes a pair of transform circuits 176A and 176B, a branch predictor 190, and a multiplexor (mux) 192.

Each of transform circuits 176A and 176B may be similar to transform circuit 176, and are coupled to receive the register indexes from stack transform storage 174 and free list 172, as described above for transform circuit 176. Additionally, translate control circuit 170 is configured to generate a sequential fetch address and a non-sequential fetch address in the present embodiment. More particularly, if translate control circuit 170 detects a branch instruction in the sequential instructions received during a clock cycle, translate control circuit 170 may generate the sequential address to the branch instruction and the target (non-sequential) address of the branch instruction. Both addresses may be transmitted to fetch unit 152 for fetching. Fetch unit 152 may provide the sequential instructions and the non-sequential instructions to predecode unit 160, which may predecode both sets of instructions concurrently. Predecode unit 160 may provide the predecoded instructions and stack change information to decode unit 162, which may decode each set of instructions concurrently. Decode unit 162 may provide the sequential instructions fetched in response to the sequential address to transform circuit 176A, and the non-sequential instructions fetched in response to the non-sequential address to transform circuit 176B. Each transform circuit 176A and 176B assigns register indexes as described above, and generates a final transform. The outputs of each transform circuit 176A and 176B are provided to mux 192, which is controlled by branch predictor 190.

In addition to generating two fetch addresses in response to a branch instruction, translate control circuit 170 may inform branch predictor 190 of the branch instruction. Branch predictor 190 may employ any suitable branch prediction algorithm. Branch predictor 190 predicts the branch instruction, and selects the sequential instructions (from transform circuit 176A) if the prediction is not-taken and the non-sequential instructions (from transform circuit 176B) if the prediction is taken. The selected instructions are provided to write unit 156, and stack transform storage 174 and free list 172 are updated according to the final transform corresponding to the

selected instructions.

Since the translation subsequent to a predicted branch instruction is speculative, translate unit 154 may be configured to store a shadow copy of the free list and stack transform for recovery if the prediction is incorrect. Translate control circuit 170 may update the status register in response to detecting the branch instruction (and thus CPU 12 may be interrupted to execute the translated code), as described above, while speculatively translating additional code. Additionally, the address of the first instruction in the predicted path would be the source address stored by JVM 40 into source address register 26 during the next activation of code translator 22. Thus, translate control circuit 170 may be configured to store the address of the first predicted instruction and to compare that address to the address stored in source address register 26 upon the next activation (via the "go" command being stored in control register 30). If the addresses match, the speculative translation was correct and may continue. If the addresses do not match, the speculative translation was incorrect and the shadow copies of the free list and stack transform may be copied back into free list 172 and stack transform storage 174. Translation down the correct path may then be performed.

As illustrated in Fig. 10, fetch unit 152 may include a cache 200, muxes 194, 196, and 198, and a fetch control circuit 160. Cache 200 may include two ports in this embodiment. The first port, labeled IA1 in Fig. 10, may be used to fetch the sequential addresses (first the source address from source address register 26, then the sequential addresses from translate control circuit 170, as selected through mux 194). The second port, labeled IA2 in Fig. 10, may be used to fetch the non-sequential addresses or a prefetch address from fetch control circuit 160 (which may employ any suitable prefetch algorithm), as selected through mux 196. Fetch control circuit 160 may provide selection controls to both muxes 194 and 196. Generally, if a "go" command is received from control register 30, fetch control unit 160 may select source address register 26 through mux 194, otherwise fetch control unit 160 may select the sequential fetch address provided by translate unit 154. Similarly, if a non-sequential fetch address is provided from translate unit 154, fetch control circuit 160 may be configured to select the non-sequential fetch address through mux 196. Otherwise, the prefetch address may be selected.

Miss information corresponding to both input addresses is provided to fetch control circuit 160, and fetch control circuit 160 may control mux 198 for providing a fetch request and address to PCI interface unit 150. If one of the addresses on the input address ports to cache 200 is a miss, that address may be selected via mux 198. If both addresses are a miss, the address on IA1 is selected. If both addresses are a hit in cache 200, the prefetch address may be selected.

In one embodiment, the first port (IA1) on cache 200 is a read-only port while the second port (IA2) is read-write. Thus, the source instructions from PCI interface 150 may be provided to an input data port corresponding to IA2. Thus, if a miss is detected on the first port, the address is re-presented on the second port when the corresponding instructions are provided for storage in cache 200. It is noted that cache 200 is optional, and if not implemented fetch control unit 160 may provide one or more fetch addresses to PCI interface 150 for fetching.

Turning next to Fig. 15, a flowchart illustrating operation of one embodiment of decode unit 162 for decoding source instructions is shown. Other embodiments are possible and contemplated. While the steps shown are illustrated in a particular order for ease of understanding, any suitable order may be used. Particularly, various steps shown may be performed in parallel by combinatorial logic within decode unit 162. Decode unit 162 may

perform the steps shown in Fig. 15 each cycle that source instructions are provided by predecode unit 160 for decoding.

Decode unit 162 determines if additional registers are needed in the register pool used by code translator 22 to store stack items (decision block 200). For example, decode unit 162 may compare the spill count in spill count register 168 to the cumulative stack pointer modification in VSP register 166. If the cumulative stack pointer modification indicates that the number of pushes minus the number of pops exceeds the number of registers in the register pool (or exceeds a threshold value near the number of registers in the register pool), decode unit 162 may generate target instructions to store the values in additional registers to scratchpad memory 50 so that the registers may be added to the register pool (step 202). Alternatively, decode unit 162 may cause an interrupt to be asserted if the number of allocated registers reaches a predetermined threshold, to allow software to spill the registers to the operand stack and thus free them for use for new stack operands.

Decode unit 162 generates target instructions for the source instructions, as described above (step 204). Additionally, decode unit 162 updates VPC register 167 and VSP register 166 with the cumulative effects of the source instructions (step 206).

Decode unit 162 further determines, from the stop fetch signal from stack to register transform unit 164, whether or not the translation is complete (decision block 208). If the translation is complete, decode unit 162 generates target instructions which adjust the PC register with the cumulative PC modification recorded in VPC register 167, adjust the stack pointer register with the cumulative stack pointer modification recorded in VSP register 166, and restore registers allocated to the register pool by decode unit 162 (step 210).

Turning now to Fig. 16, an exemplary code sequence 220 is shown. Code sequence 220 may be generated by an embodiment of decode unit 162 according to the flowchart shown in Fig. 15. Decode unit 162 generates target instructions corresponding to the source instructions (e.g. target instructions 222) until the translation is determined to be complete (e.g. due to exception, translating a number of instructions equal to the translation limit, detecting an instruction which is not translated by code translator 22, etc.). Upon detecting that the translation is complete, decode unit 162 may generate target instructions to adjust the PC and stack pointer registers and to restore registers allocated to the register pool by decode unit 162.

More particularly, decode unit 162 may generate a target instruction 224 to update the PC register and a target instruction 226 to update the stack pointer register. In the embodiment shown, instruction 224 may be an add instruction having the PC register as a destination register and as a source register, and having an immediate field carrying the cumulative PC modification (VPC) from VPC register 167. Similarly, instruction 226 may be an add instruction having the stack pointer register as a destination register and as a source register, and having an immediate field carrying the cumulative stack pointer modification (VSP) from VSP register 166. Thus, the cumulative effects of the translated instructions on each of the PC and stack pointer registers may be reflected in the registers of CPU 12 used by the JVM to store the PC and stack pointer values.

Target instructions 228 may be instructions which restore the registers allocated to the register pool during the translation. More particularly, each register which is currently storing a stack item may be restored using a store instruction (to store the stack item in the register to the operand stack) and a load instruction (to load the saved value of the register from the scratch area). Potentially, a register no longer contains useful data and only a load is generated. Each store instruction may store a register value to various offsets from the stack pointer register (e.g.

the first store instruction to offset 0, the second store instruction to offset 4, etc.). Thus, the register storing the top of stack item is stored to the top of the stack by the first store instruction, the register storing the second to the top of stack item is stored to the second to the top of stack entry by the second store instruction, etc. In the embodiment shown, stack items are 32 bits, although other embodiments may employ different sizes. Additionally, the stack change information provided by decode unit 162 with the store instructions may indicate no stack change (no pushes and no pops), while the stack change information corresponding to the load instructions may indicate a pop, so that the next store instruction may receive the next register index down from the top of stack from the stack transform as the source register. For registers which are on the free list (and thus are not currently storing stack items), a load instruction to load the saved value from the scratch area may be generated (with the corresponding stack change information indicating no pushes or pops).

It is noted that, if registers are reserved for the register pool, rather than allocated, the load instructions may be eliminated from target instructions 228. Additionally, the stack change information corresponding to the store instructions may indicate a pop, so that the next store instruction receives the next register index down from the stack transform, as described above.

Finally, code sequence 220 may conclude with a return instruction to the JVM (reference numeral 230).

Turning next to Fig. 17, a flowchart illustrating operation of one embodiment of decode unit 162 for decoding a source conditional branch instruction is shown. Other embodiments are possible and contemplated. While the steps shown are illustrated in a particular order for ease of understanding, any suitable order may be used. Particularly, various steps shown may be performed in parallel by combinatorial logic within decode unit 162. Additionally, Fig. 18 illustrates an exemplary target code sequence 250 which may be generated by the embodiment of decode unit 162 shown in Fig. 17.

The embodiment illustrated by Figs. 17 and 18 may be used if translation beyond a conditional branch in the source code sequence is speculatively performed by code translator 22. The embodiment illustrated in Figs. 17 and 18 may be an alternative embodiment for handling conditional branches than the embodiment shown in Figs. 10 and 11. By employing the embodiment shown in Figs. 17 and 18, the restoration of registers from the register pool and the adjustment of the PC register and stack pointer register may be delayed until a return to the JVM is actually performed. For example, in the case of a conditional branch, code translator 22 may predict a direction for the conditional branch (e.g. not taken). If not taken is predicted, decode unit 162 may operate as shown in Figs. 17 and 18. Alterations if taken is predicted are described below.

As shown in Fig. 17, decode unit 162 may generate a target conditional branch instruction which is checking for the logical opposite of the condition checked for by the source conditional branch instruction (step 240). For example, if the source conditional branch instruction is checking for a condition of greater than, the logical opposite is less than or equal. If the source conditional branch instruction is checking for equal, the logical opposite is not equal, etc. In other words, if the source conditional branch instruction results in taken, the target conditional branch instruction checking for the logically opposite condition is not taken. Similarly, if the source conditional branch instruction results in not taken, the target conditional branch instruction is taken. The target address of the target conditional branch instruction generated in step 240 is explained in more detail below. The target conditional branch instruction generated in response to step 240 is illustrated in code sequence 250 as instruction 252.

Decode unit 162 generates the target instructions to adjust the PC and stack pointer registers, based on the values in the VPC and VSP registers 167 and 166, respectively (step 242). Additionally, target instructions to restore the registers allocated to the register pool are generated. Step 242 may be similar to step 210 described above. Accordingly, instructions 224, 226, and 228 from Fig. 16 are illustrated in Fig. 18 as well. Finally, decode unit 162 generates a second target branch instruction (step 244). The second target branch instruction may be a return instruction to the JVM to determine if the source conditional branch instruction's target address corresponds to a translated code sequence or to determine if translation is to be initiated at the source conditional branch instruction's target address. Alternatively, the second target branch instruction may be a second target conditional branch instruction checking for the same condition as the source conditional branch instruction and having a target address of instructions translated from the source instructions at the target of the source conditional branch instruction. The second target branch instruction is illustrated in code sequence 250 as instruction 254.

In the case illustrated in Figs. 17 and 18, the source conditional branch instruction is predicted not taken. If the prediction is correct, the target conditional branch instruction generated at step 240 is taken. Accordingly, the target address of the target conditional branch instruction is set to bypass the instructions which adjust the PC and stack pointer and restore the registers in the register pool and further to bypass the second target branch instruction. Thus, the target address of the target conditional branch instruction generated in step 240 is the address of the instruction succeeding the second target branch instruction generated in step 244 (see arrow 256 in Fig. 18). Thus, in the present embodiment, the target address may be relative to the target conditional branch instruction generated in step 240 and may be the restore count plus 3 instructions (the two adjust instructions and the second target branch instruction).

Thus, if the prediction is correct, adjustment of the PC and stack pointer registers and restoration of the registers allocated to the register pool may be delayed, and execution may continue with additional target instructions 258, generated from source instructions sequential to the source conditional branch instruction (translated after decode unit 162 performs the translation for the source conditional branch instruction as illustrated in Fig. 17). If the prediction is incorrect, the adjustment and the restoration may be performed.

On the other hand, if the conditional branch instruction is predicted taken, target conditional branch instruction 252 may be generated to check for the same condition as the source conditional branch instruction. The target address of target conditional branch instruction 252 may remain the same as shown in Fig. 18. Additionally, in the predicted taken case, target instructions 258 may comprise instructions translated from source instructions at the target address of the source conditional branch instruction.

It is noted that the above description refers to PC and stack pointer registers maintained by the JVM for a Java code sequence. These registers may be predetermined to be in certain registers of the register set employed by CPU 12 (and thus the register indexes for these registers may be predetermined for code translator 22). Alternatively, code translator 22 may include a configuration register (not shown) which may be programmed with the register indexes of each register.

Numerous variations and modifications will become apparent to those skilled in the art once the above disclosure is fully appreciated. It is intended that the following claims be interpreted to embrace all such variations and modifications.

WHAT IS CLAIMED IS:

1. An apparatus comprising:

a storage configured to store a plurality of register indexes, said plurality of register indexes identifying a plurality of registers storing a plurality of stack items comprising a top portion of a stack; and

a transform circuit coupled to said storage and coupled to receive one or more instructions and corresponding stack change information, wherein said transform circuit is configured to assign one or more of said plurality of register indexes to each of one or more source operands of said one or more instructions responsive to said stack change information.

2. The apparatus as recited in claim 1 wherein said transform circuit is configured to update said storage responsive to said stack change information.

3. The apparatus as recited in claim 2 further comprising a free list storage configured to store a second plurality of register indexes, said second plurality of register indexes being free for assignment to destination operands of said one or more instructions.

4. The apparatus as recited in claim 3 wherein said transform circuit is configured to update said storage with one or more of said second plurality of register indexes responsive to said one or more instructions consuming said one or more of said second plurality of register indexes.

5. The apparatus as recited in claim 3 wherein a first instruction of said one or more instructions has a destination operand if said first instruction pushes a result onto said stack.

6. The apparatus as recited in claim 1 further comprising a second transform circuit coupled to receive one or more instructions from a non-sequential path and corresponding stack change information responsive to a preceding branch instruction, wherein said second transform circuit is coupled to said storage, and wherein said second transform circuit is configured to assign one or more of said plurality of register indexes to each of one or more source operands of said one or more instructions from said non-sequential path responsive to said stack change information corresponding to said one or more instructions from said non-sequential path.

7. The apparatus as recited in claim 6 further comprising a selection circuit coupled to said transform circuit and said second transform circuit, wherein said selection circuit is configured to select an output set of one or more instructions from said transform circuit and said second transform circuit responsive to a branch prediction corresponding to said preceding branch instruction.

8. A method comprising:

receiving one or more instructions and corresponding stack change information; and

assigning one or more register indexes to one or more source operands of said one or more instructions,

said one or more register indexes read from a storage and said one or more register indexes indicative of registers storing one or more stack items forming a top portion of a stack.

9. The method as recited in claim 8 further comprising updating said storage responsive to said stack change information.

10. The method as recited in claim 9 further comprising assigning destination register indexes from a free list of register indexes.

11. The method as recited in claim 10 wherein said updating comprises storing said destination register indexes from said free list into said storage.

12. The method as recited in claim 8 further comprising:
receiving one or more instructions and corresponding stack change information from an alternate path responsive to a preceding branch instruction; and
assigning one or more register indexes to one or more source operands of said one or more instructions from said alternate path responsive to said stack change information corresponding to said alternate path.

13. The method as recited in claim 12 further comprising selecting one of one or more instructions from said alternate path or said one or more instructions and updating said stack transform storage responsive to said selecting.

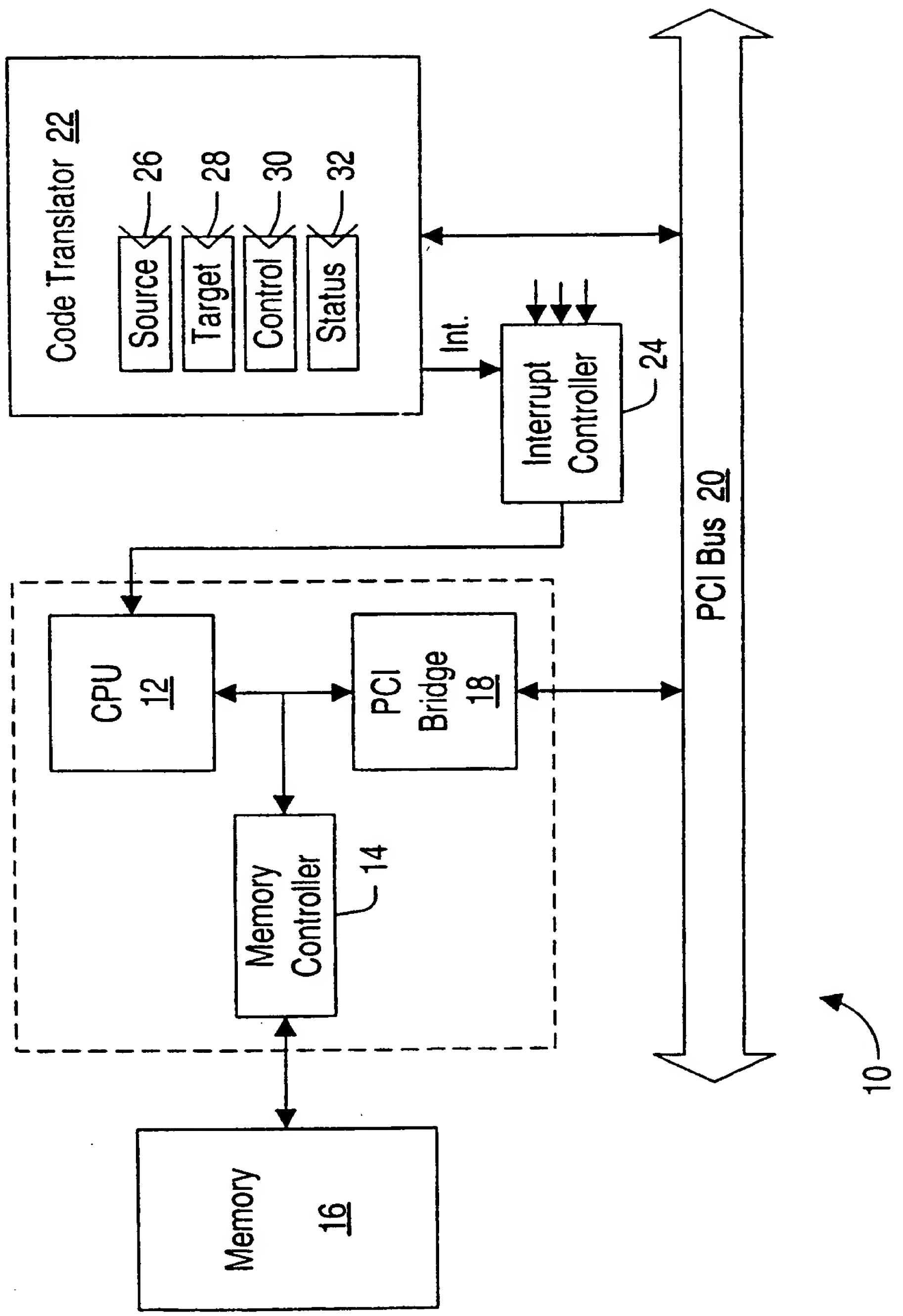

Code Translator 22
Source
Target
Control
Status
26
28
30
32
Int.
Interrupt Controller
24
CPU 12
PCI Bridge 18
Memory Controller
14
Memory 16
PCI Bus 20
10

Fig. 1

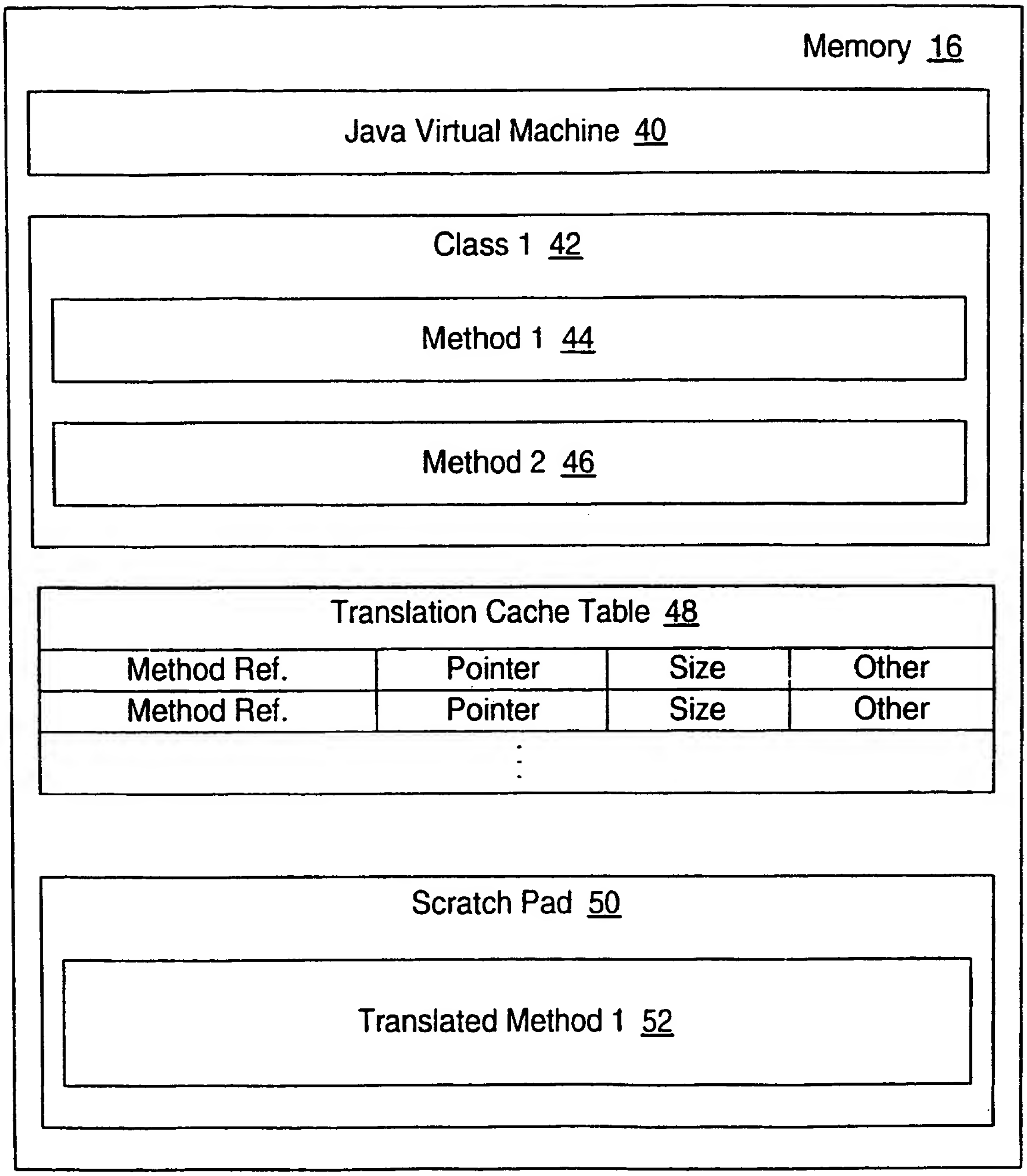

Memory 16
Java Virtual Machine 40
Class 1 42
Method 1 44
Method 2 46
Translation Cache Table 48
Method Ref.
Pointer
Size
Other
Method Ref.
Pointer
Size
Other
Scratch Pad 50
Translated Method 1 52

Fig. 2

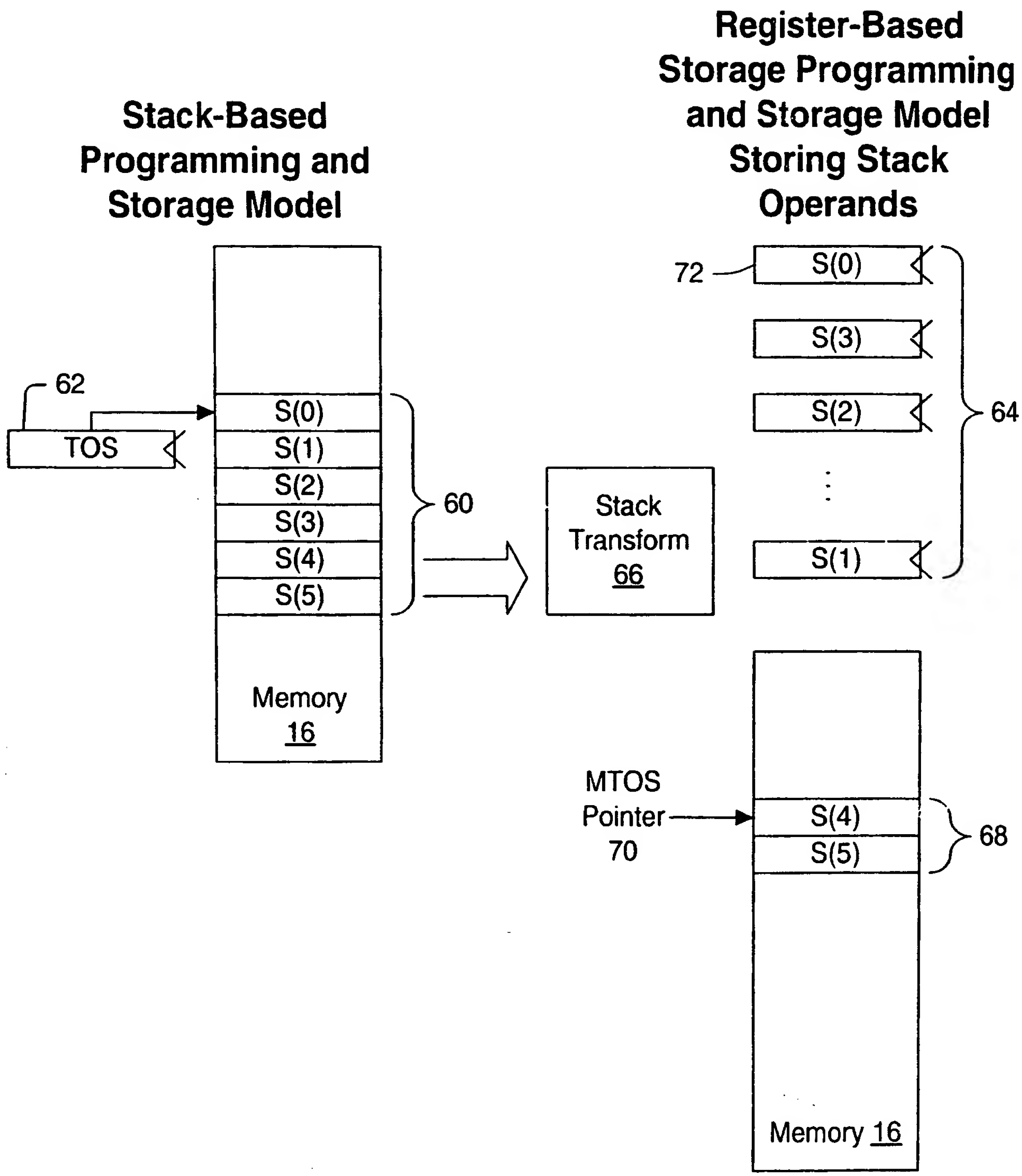

Stack-Based
Programming and
Storage Model
Register-Based
Storage Programming
and Storage Model
Storing Stack
Operands
72
S(0)
S(3)
S(2)
S(1)
64
62
TOS
S(0)
S(1)
S(2)
S(3)
S(4)
S(5)
60
Stack
Transform
66
Memory
16
MTOS
Pointer
70
S(4)
S(5)
68
Memory 16

Fig. 3

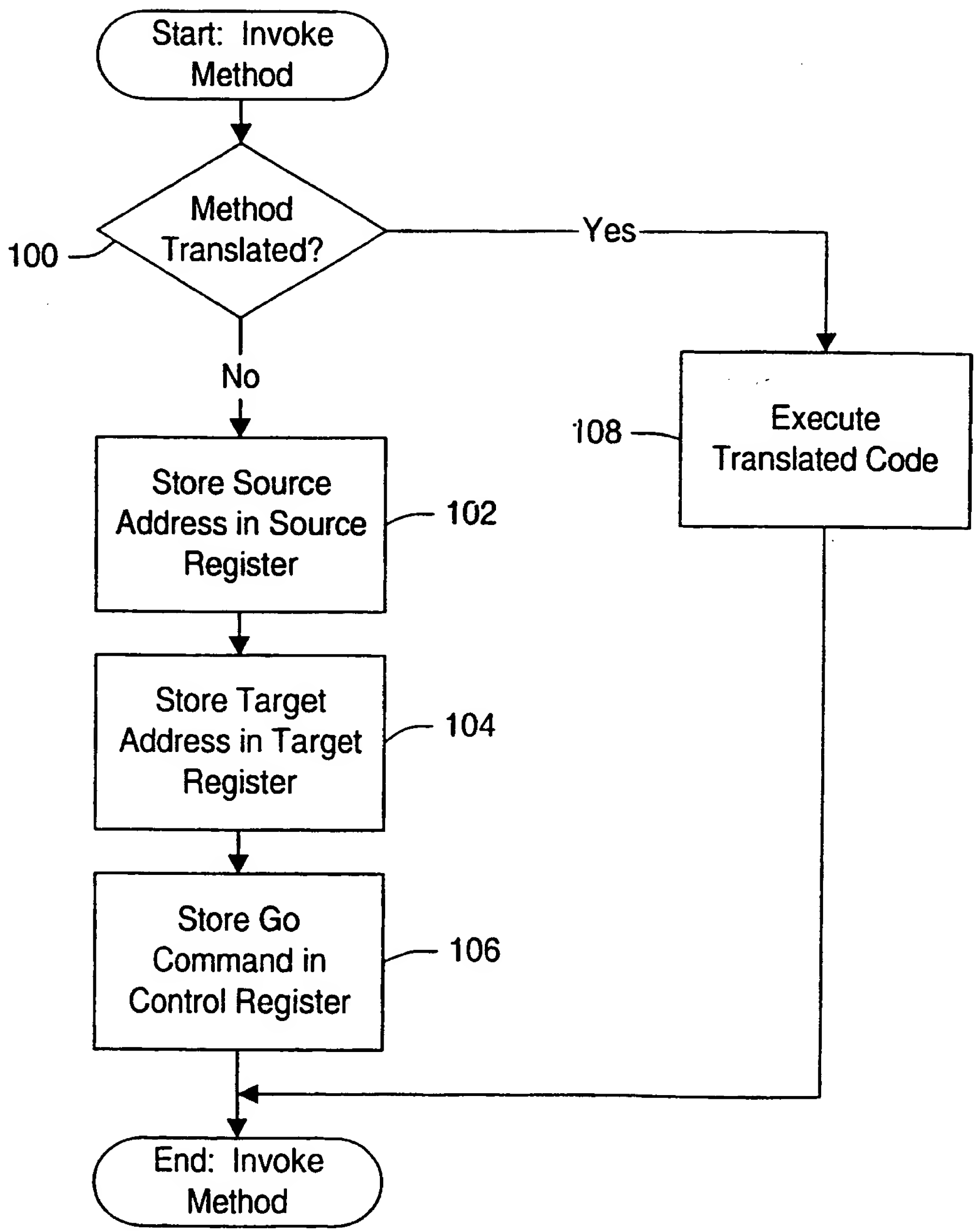

Start: Invoke Method
Method Translated?
100
Yes
No
Execute Translated Code
108
Store Source Address in Source Register
102
Store Target Address in Target Register
104
Store Go Command in Control Register
106
End: Invoke Method

Fig. 4

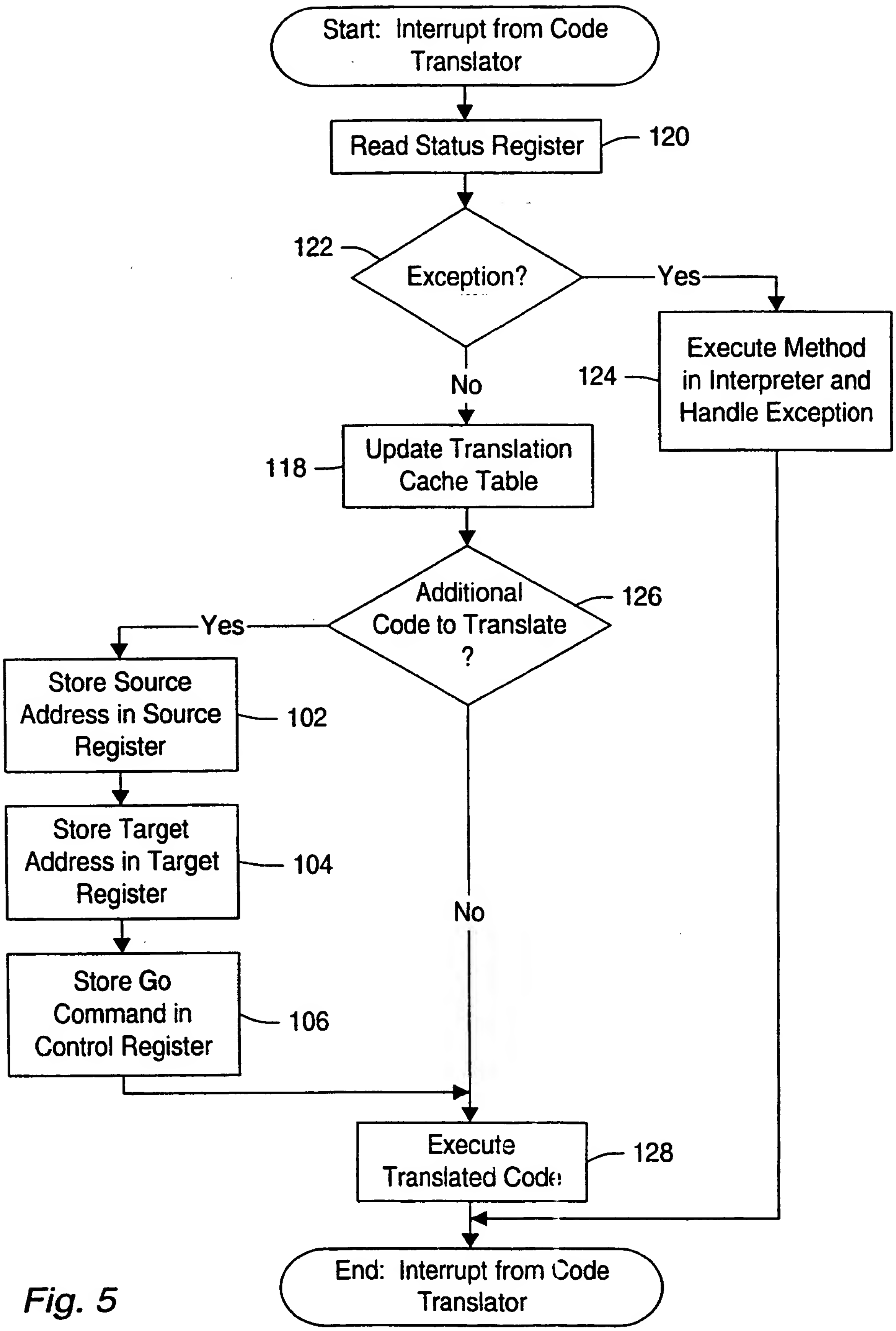
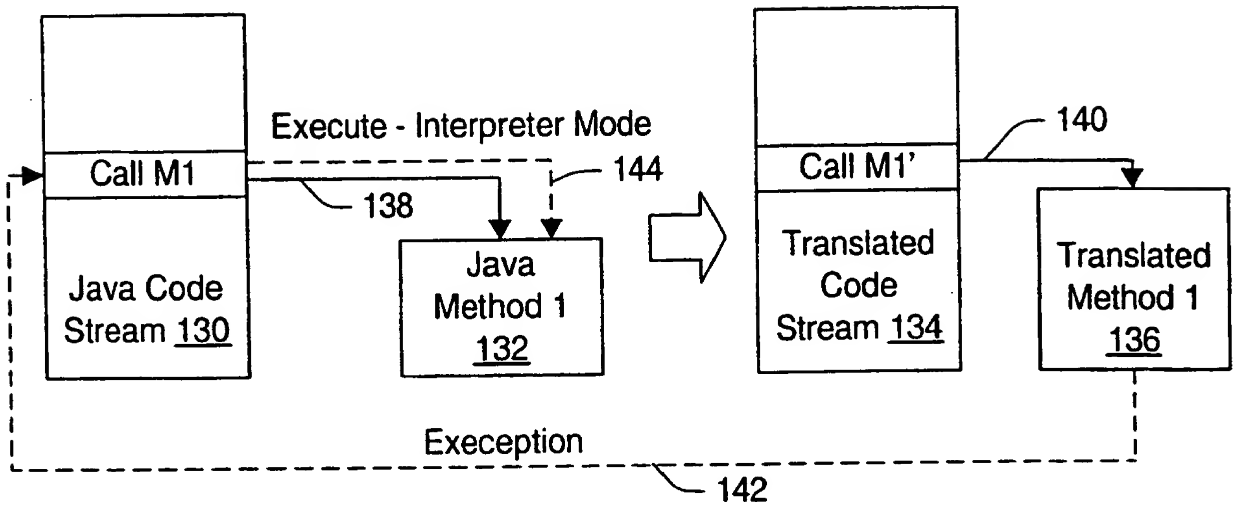
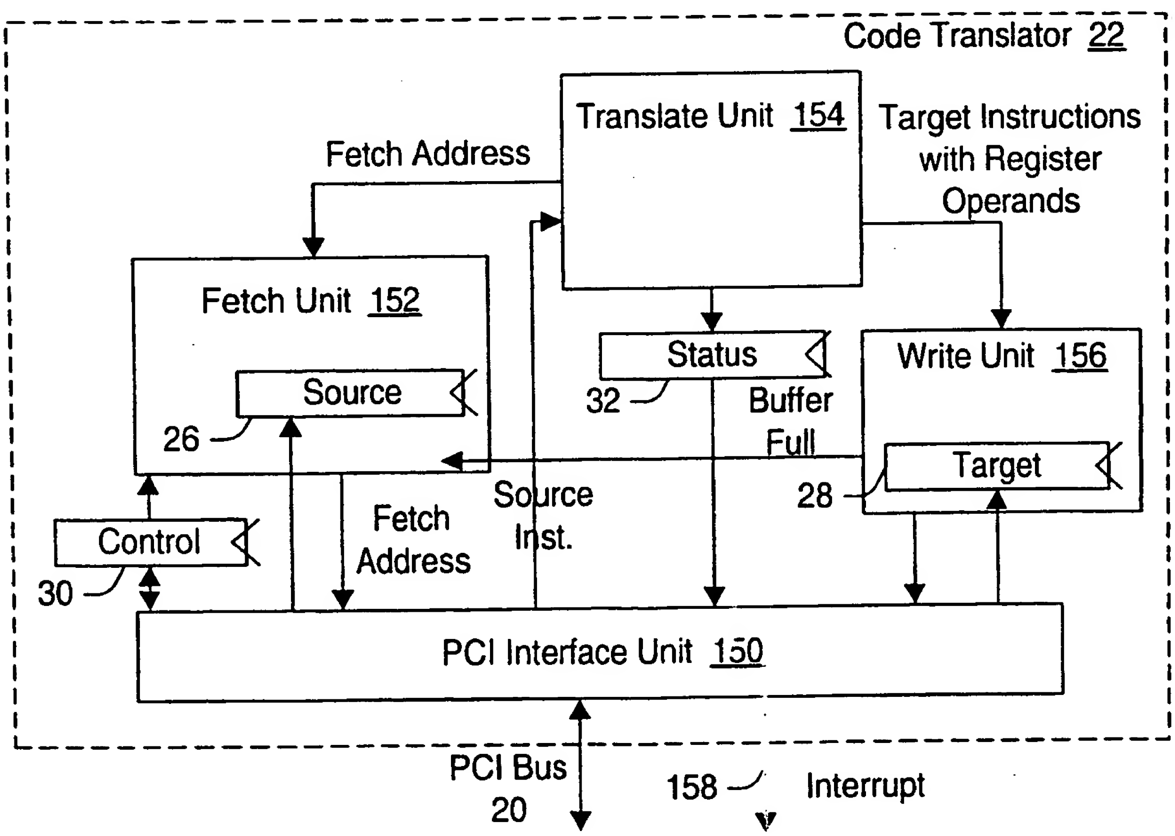

Start: Interrupt from Code Translator
Read Status Register
120
122
Exception?
Yes
Execute Method in Interpreter and Handle Exception
124
No
Update Translation Cache Table
118
Additional Code to Translate ?
126
Yes
Store Source Address in Source Register
102
Store Target Address in Target Register
104
Store Go Command in Control Register
106
No
Execute Translated Code
128
End: Interrupt from Code Translator

Fig. 5



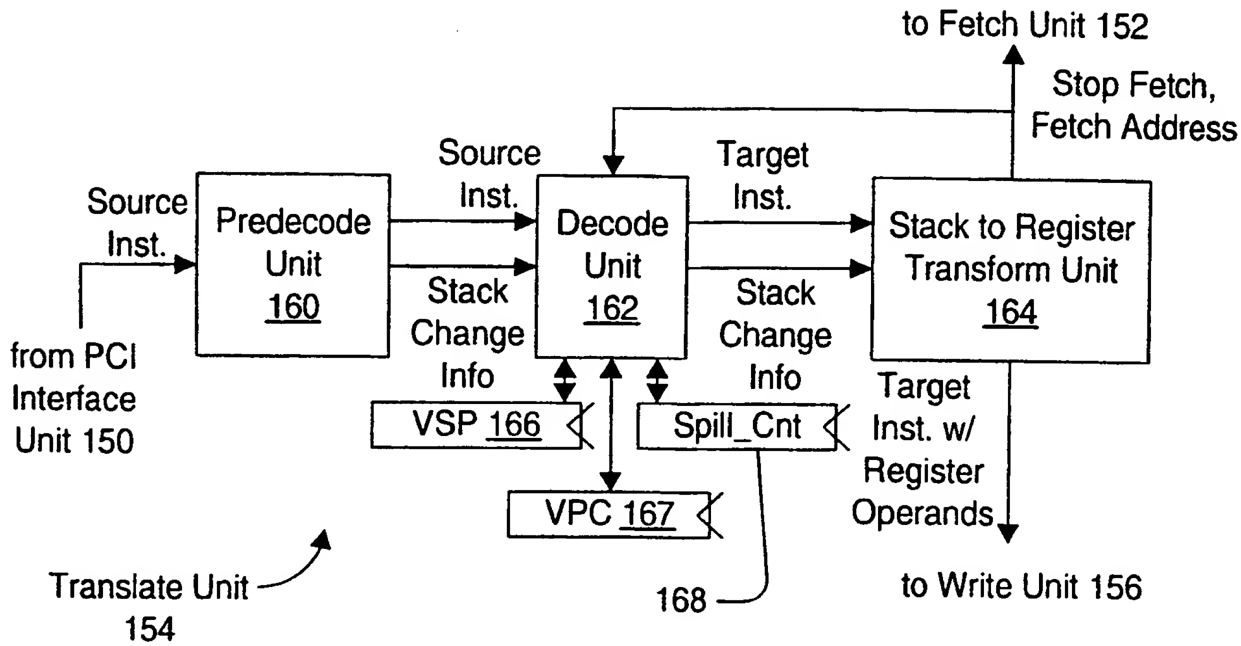
Execute - Interpreter Mode
Call M1
138
144
Java Code Stream 130
Java Method 1 132
Call M1'
140
Translated Code Stream 134
Translated Method 1 136
Exeception
142

Fig. 6



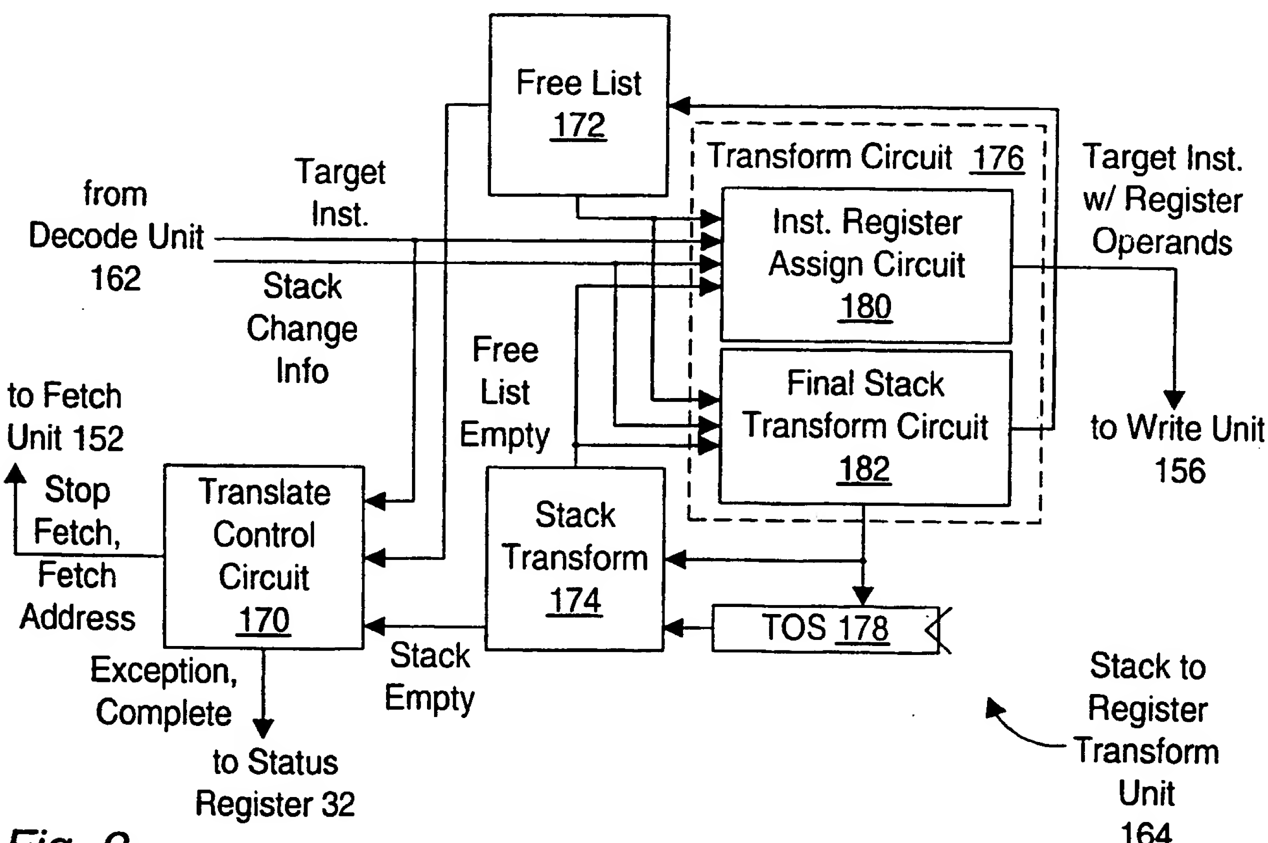
Code Translator 22
Translate Unit 154
Fetch Address
Target Instructions with Register Operands
Fetch Unit 152
Source
26
Status
32
Buffer Full
Write Unit 156
Target
28
Control
30
Fetch Address
Source Inst.
PCI Interface Unit 150
PCI Bus 20
158
Interrupt

Fig. 7



to Fetch Unit 152
Stop Fetch, Fetch Address
Source Inst.
Target Inst.
Source Inst.
Predecode Unit 160
Decode Unit 162
Stack to Register Transform Unit 164
Stack Change Info
Stack Change Info
from PCI Interface Unit 150
VSP 166
Spill_Cnt
Target Inst. w/ Register Operands
VPC 167
Translate Unit 154
168
to Write Unit 156

Fig. 8



Free List 172
Transform Circuit 176
Target Inst. w/ Register Operands
from Decode Unit 162
Target Inst.
Inst. Register Assign Circuit 180
Stack Change Info
Free List Empty
Final Stack Transform Circuit 182
to Fetch Unit 152
to Write Unit 156
Stop Fetch, Fetch Address
Translate Control Circuit 170
Stack Transform 174
TOS 178
Exception, Complete
Stack Empty
to Status Register 32
Stack to Register Transform Unit 164

Fig. 9

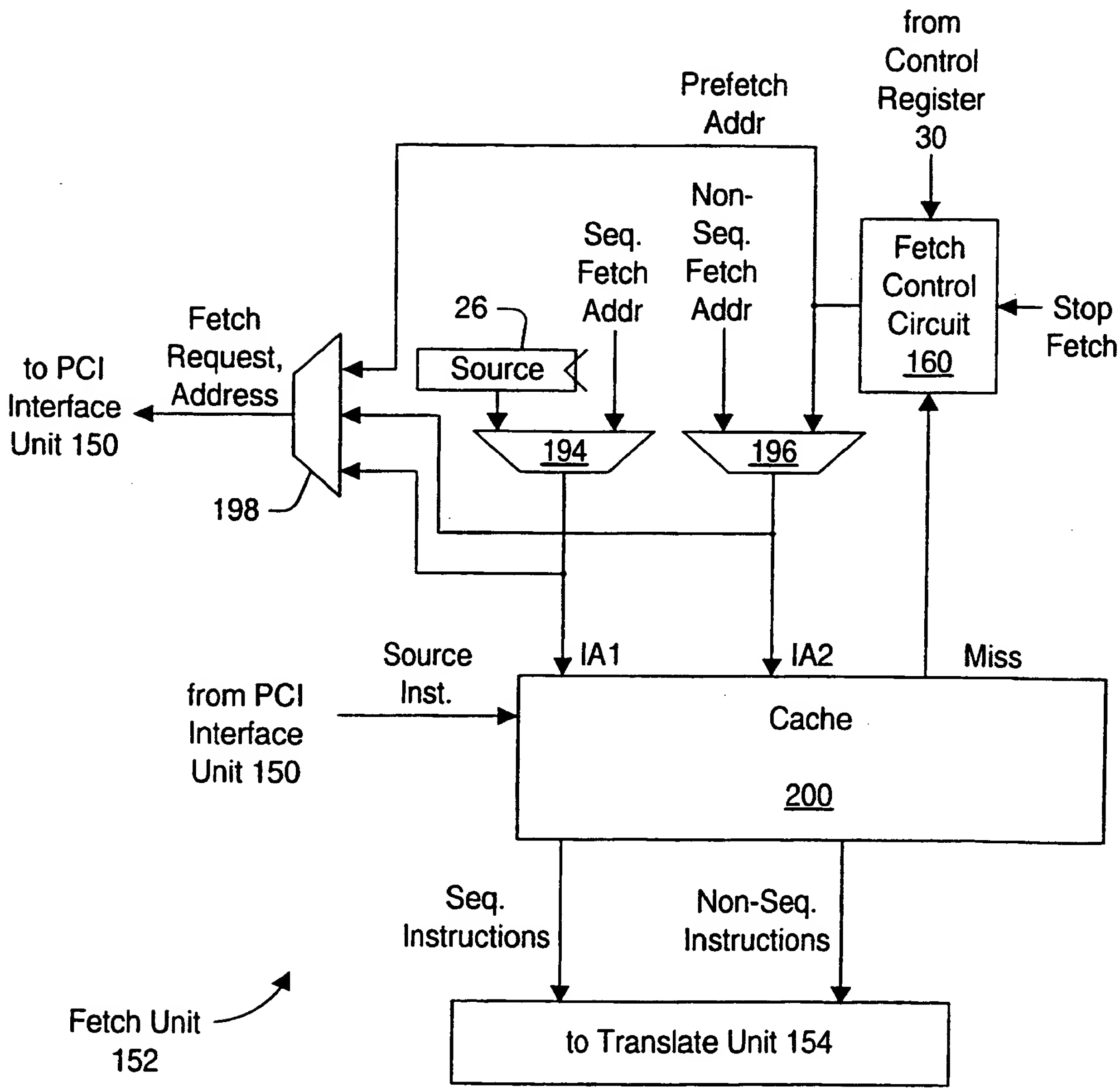

from
Control
Register
30
Prefetch
Addr
Non-
Seq.
Fetch
Addr
Seq.
Fetch
Addr
26
Source
Fetch
Control
Circuit
160
Stop
Fetch
Fetch
Request,
Address
to PCI
Interface
Unit 150
194
196
198
IA1
IA2
Miss
Source
Inst.
from PCI
Interface
Unit 150
Cache
200
Seq.
Instructions
Non-Seq.
Instructions
to Translate Unit 154
Fetch Unit
152

Fig. 10

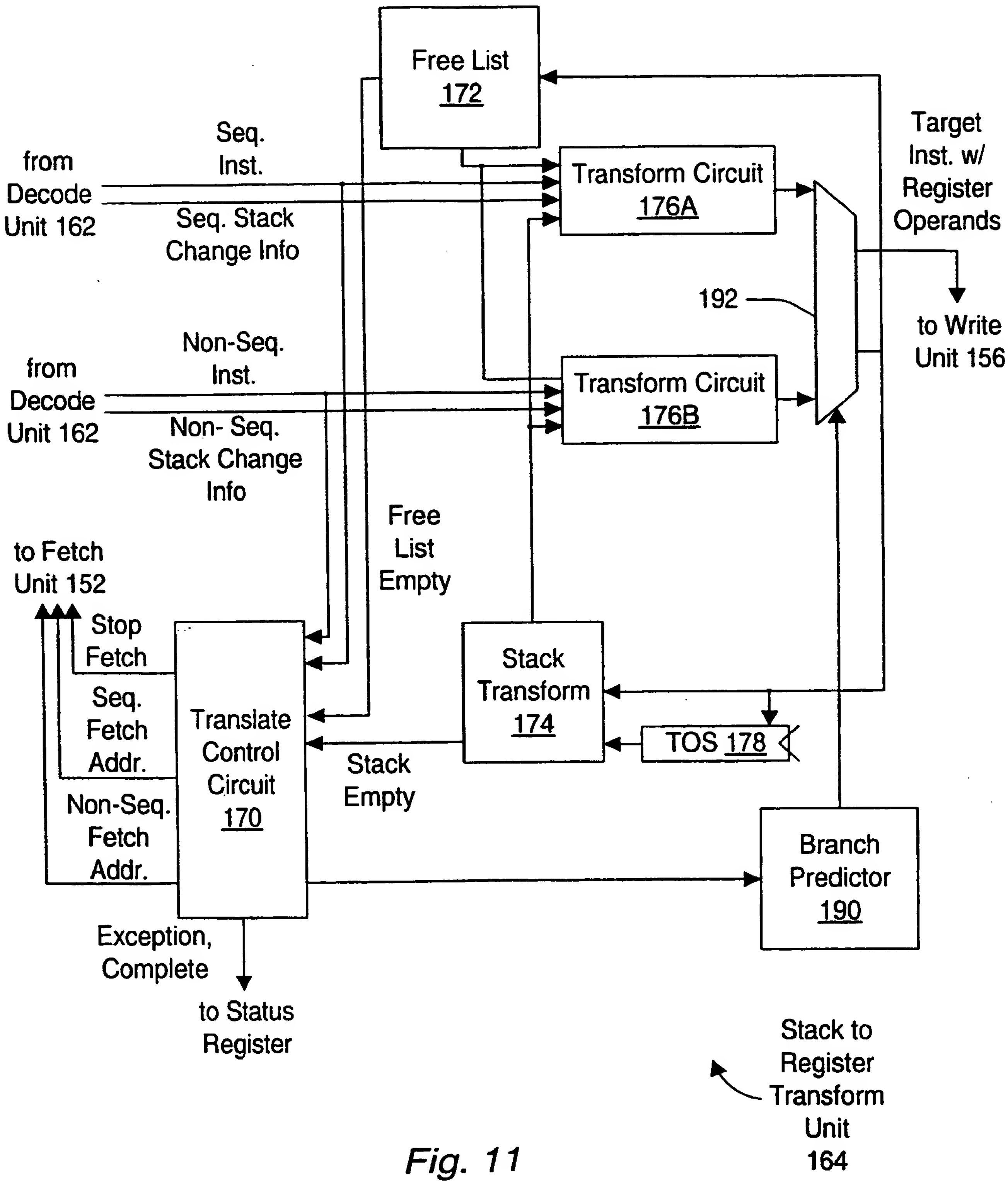

Free List
172
from
Decode
Unit 162
Seq.
Inst.
Seq. Stack
Change Info
Transform Circuit
176A
Target
Inst. w/
Register
Operands
192
to Write
Unit 156
from
Decode
Unit 162
Non-Seq.
Inst.
Non- Seq.
Stack Change
Info
Transform Circuit
176B
Free
List
Empty
to Fetch
Unit 152
Stop
Fetch
Seq.
Fetch
Addr.
Non-Seq.
Fetch
Addr.
Translate
Control
Circuit
170
Stack
Transform
174
TOS 178
Stack
Empty
Branch
Predictor
190
Exception,
Complete
to Status
Register
Stack to
Register
Transform
Unit
164

Fig. 11

Instruction 0		Instruction 1		Instruction 2	
Push	Pop	Push	Pop	Src0	Src1
0	0	0	0	S[0]	S[1]
0	1	0	0	S[1]	S[2]
1	0	0	0	F[0]	S[0]
0	0	0	1	S[1]	S[2]
0	0	1	0	F[0]	S[0]
0	1	0	1	S[2]	S[3]
0	1	1	0	F[0]	S[1]
1	0	0	1	S[0]	S[1]
1	0	1	0	F[1]	F[0]

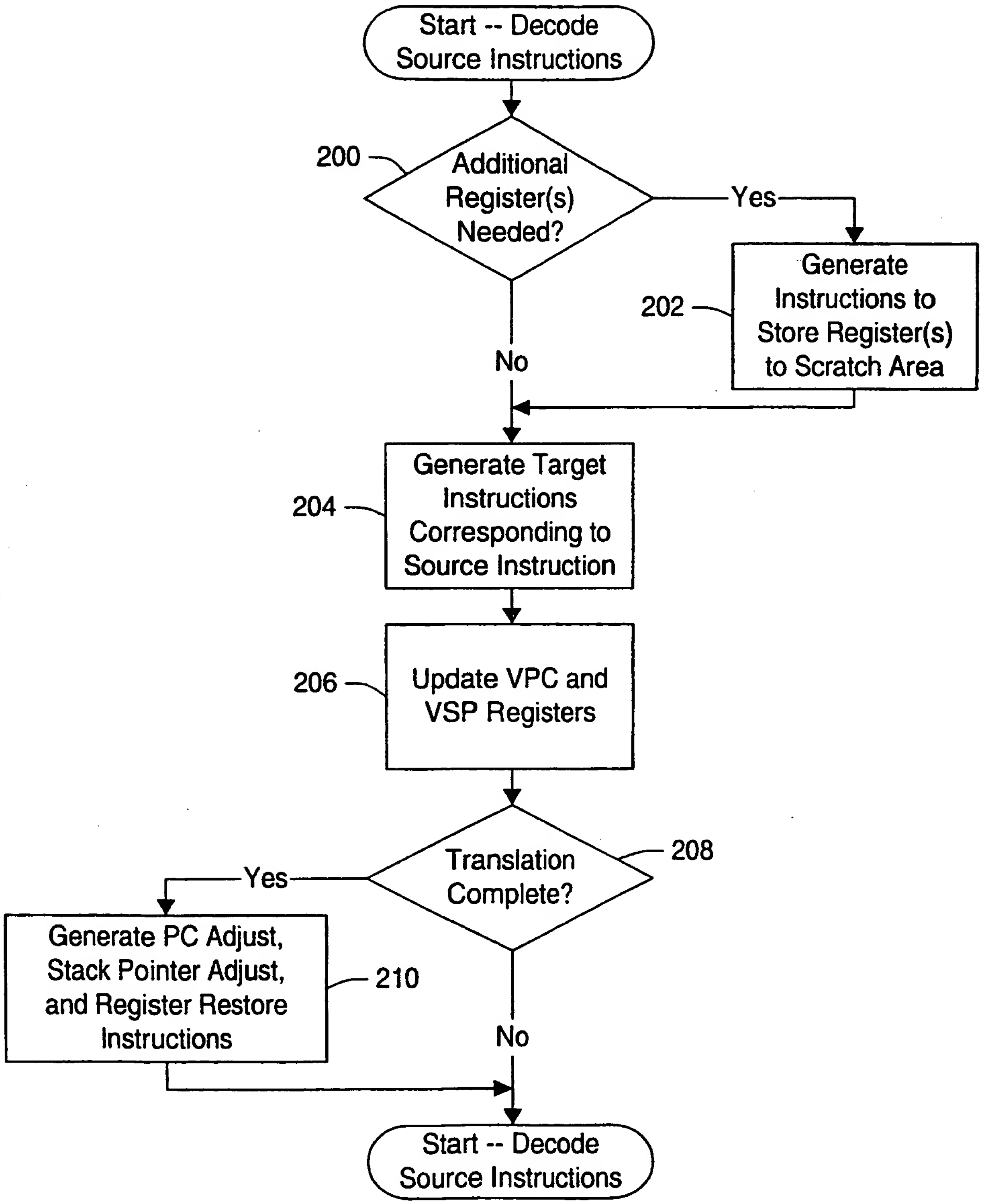
Fig. 12

Instruction 0		Instruction 1		Instruction 2		Resulting Stack Transform
Push	Pop	Push	Pop	Push	Pop	
0	0	0	0	0	0	S[0], S[1], S[2], S[3] ...
0	1	0	0	0	0	S[1], S[2], S[3], S[4] ...
1	0	0	0	0	0	F[0], S[0], S[1], S[2] ...
0	0	0	1	0	0	S[1], S[2], S[3], S[4] ...
0	0	1	0	0	0	F[0], S[0], S[1], S[2] ...
0	1	0	1	0	0	S[2], S[3], S[4], S[5] ...
0	1	1	0	0	0	F[0], S[1], S[2], S[3] ...
1	0	0	1	0	0	S[0], S[1], S[2], S[3] ...
1	0	1	0	0	0	F[1], F[0], S[0], S[1] ...
0	0	0	0	0	1	S[1], S[2], S[3], S[4] ...
0	1	0	0	0	1	S[2], S[3], S[4], S[5] ...
1	0	0	0	0	1	S[0], S[1], S[2], S[3] ...
0	0	0	1	0	1	S[2], S[3], S[4], S[5] ...
0	0	1	0	0	1	S[0], S[1], S[2], S[3] ...
0	1	0	1	0	1	S[3], S[4], S[5], S[6] ...
0	1	1	0	0	1	S[1], S[2], S[3], S[4] ...
1	0	0	1	0	1	S[1], S[2], S[3], S[4] ...
1	0	1	0	0	1	F[0], S[0], S[1], S[2] ...
0	0	0	0	1	0	F[0], S[0], S[1], S[2] ...
0	1	0	0	1	0	F[0], S[1], S[2], S[3] ...
1	0	0	0	1	0	F[1], F[0], S[0], S[1] ...
0	0	0	1	1	0	F[0], S[1], S[2], S[3] ...
0	0	1	0	1	0	F[1], F[0], S[0], S[1] ...
0	1	0	1	1	0	F[0], S[2], S[3], S[4] ...
0	1	1	0	1	0	F[1], F[0], S[1], S[2] ...
1	0	0	1	1	0	F[0], S[0], S[1], S[2] ...
1	0	1	0	1	0	F[2], F[1], F[0], S[0] ...

Fig. 13

Instruction 0		Instruction 1		Instruction 2		Resulting Free List
Push	Pop	Push	Pop	Push	Pop	
0	0	0	0	0	0	F[0], F[1], F[2], F[3], ..., F[7]
0	1	0	0	0	0	F[0], F[1], F[2], F[3], ..., F[7], S[0]
1	0	0	0	0	0	F[1], F[2], F[3], ..., F[7]
0	0	0	1	0	0	F[0], F[1], F[2], F[3], ..., F[7], S[0]
0	0	1	0	0	0	F[1], F[2], F[3], ..., F[7]
0	1	0	1	0	0	F[0], F[1], F[2], F[3], ..., F[7], S[0], S[1]
0	1	1	0	0	0	F[1], F[2], F[3], ..., F[7], S[0]
1	0	0	1	0	0	F[1], F[2], F[3], ..., F[7], F[0]
1	0	1	0	0	0	F[2], F[3], ..., F[7]
0	0	0	0	0	1	F[0], F[1], F[2], F[3], ..., F[7], S[0]
0	1	0	0	0	1	F[0], F[1], F[2], F[3], ..., F[7], S[0], S[1]
1	0	0	0	0	1	F[1], F[2], F[3], ..., F[7], F[0]
0	0	0	1	0	1	F[0], F[1], F[2], F[3], ..., F[7], S[0], S[1]
0	0	1	0	0	1	F[1], F[2], F[3], ..., F[7], F[0]
0	1	0	1	0	1	F[0], F[1], F[2], ..., F[7], S[0], S[1], S[2]
0	1	1	0	0	1	F[1], F[2], F[3], ..., F[7], S[0], F[0]
1	0	0	1	0	1	F[1], F[2], F[3], ..., F[7], F[0], S[0]
1	0	1	0	0	1	F[2], F[3], ..., F[7], F[1]
0	0	0	0	1	0	F[1], F[2], F[3], ..., F[7]
0	1	0	0	1	0	F[1], F[2], F[3], ..., F[7], S[0]
1	0	0	0	1	0	F[2], F[3], ..., F[7]
0	0	0	1	1	0	F[1], F[2], F[3], ..., F[7], S[0]
0	0	1	0	1	0	F[2], F[3], ..., F[7]
0	1	0	1	1	0	F[1], F[2], F[3], ..., F[7], S[0], S[1]
0	1	1	0	1	0	F[2], F[3], ..., F[7], S[0]
1	0	0	1	1	0	F[2], F[3], ..., F[7], F[0]
1	0	1	0	1	0	F[3], ..., F[7]

Fig. 14



Start -- Decode Source Instructions
200
Additional Register(s) Needed?
Yes
No
202
Generate Instructions to Store Register(s) to Scratch Area
204
Generate Target Instructions Corresponding to Source Instruction
206
Update VPC and VSP Registers
208
Translation Complete?
Yes
No
210
Generate PC Adjust, Stack Pointer Adjust, and Register Restore Instructions
Start -- Decode Source Instructions

Fig. 15

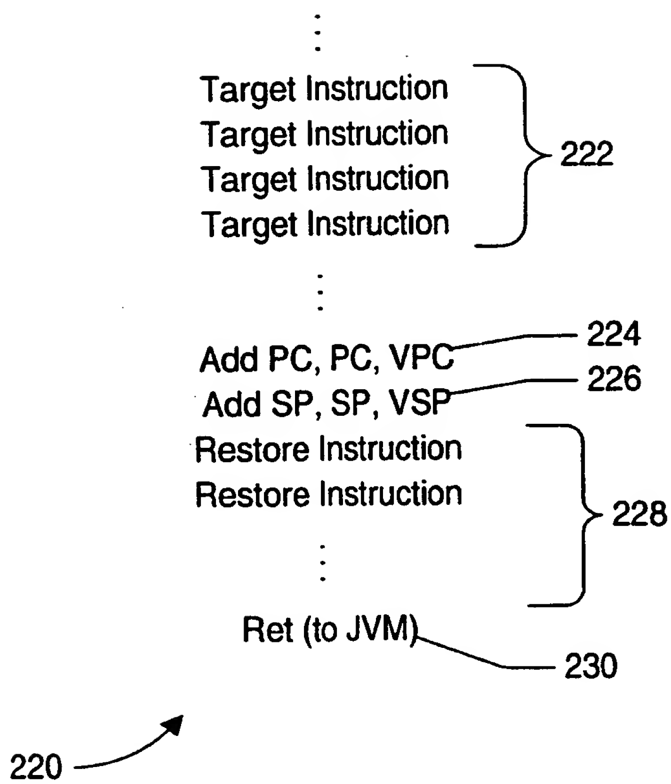

Target Instruction
Target Instruction
Target Instruction
Target Instruction
222
Add PC, PC, VPC
224
Add SP, SP, VSP
226
Restore Instruction
Restore Instruction
228
Ret (to JVM)
230
220

Fig. 16

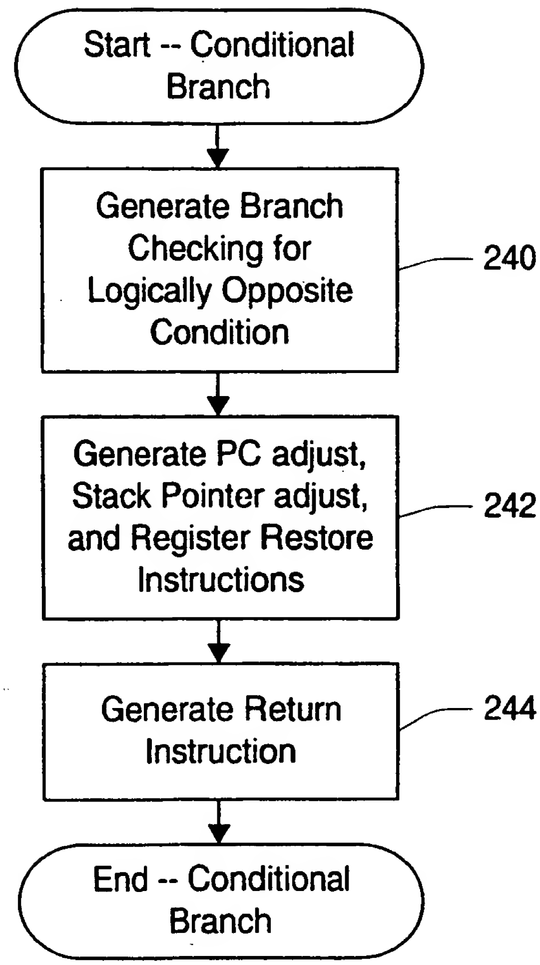

Start -- Conditional Branch
Generate Branch Checking for Logically Opposite Condition
240
Generate PC adjust, Stack Pointer adjust, and Register Restore Instructions
242
Generate Return Instruction
244
End -- Conditional Branch

Fig. 17

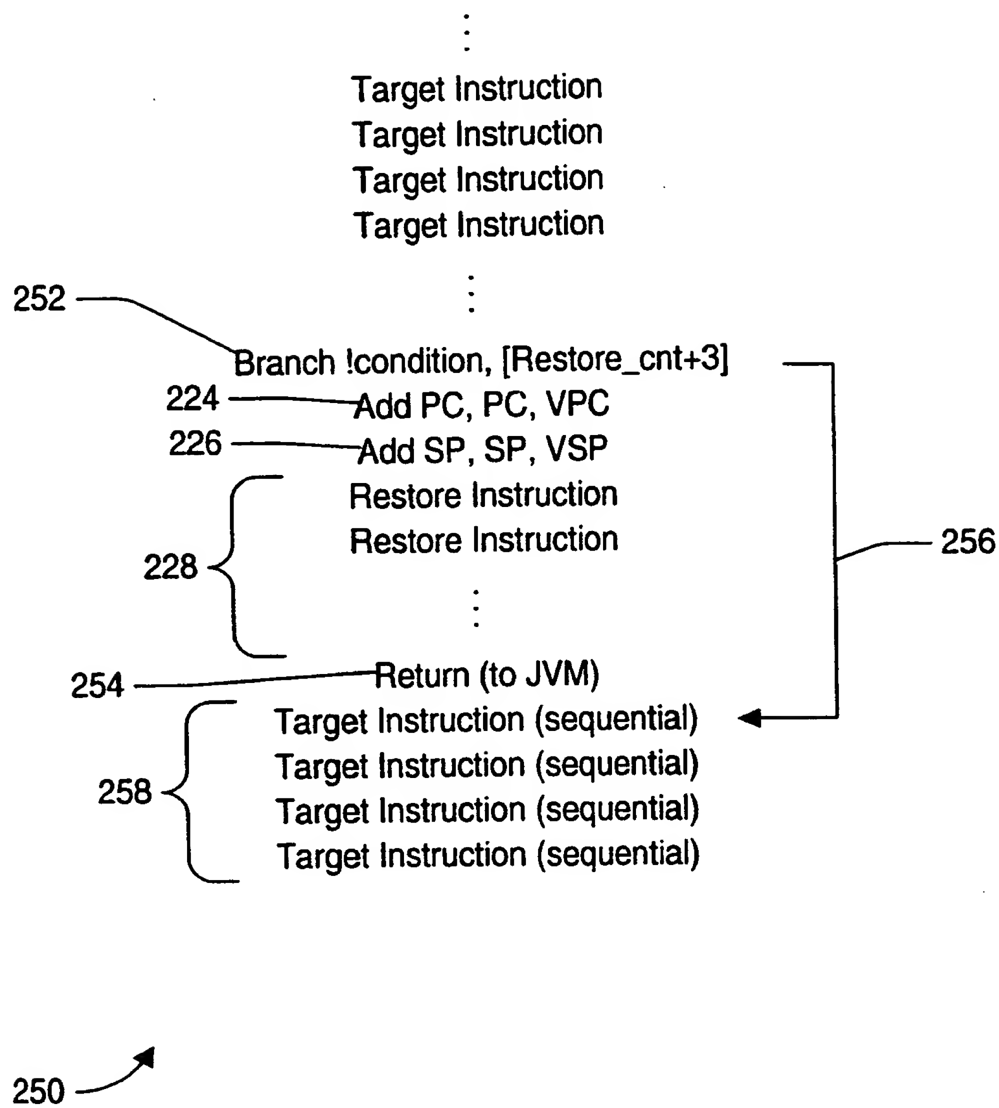

Target Instruction
Target Instruction
Target Instruction
Target Instruction
252
Branch !condition, [Restore_cnt+3]
224
Add PC, PC, VPC
226
Add SP, SP, VSP
Restore Instruction
Restore Instruction
228
256
254
Return (to JVM)
Target Instruction (sequential)
Target Instruction (sequential)
Target Instruction (sequential)
Target Instruction (sequential)
258
250

Fig. 18

INTERNATIONAL SEARCH REPORT

International Application No
PCT/L 01/04743

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 G06F9/30 G06F9/318

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

EPO-Internal

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X Y	WO 99 27439 A (SEKI HAJIME) 3 June 1999 (1999-06-03) abstract & EP 1 035 471 A (SEKI HAJIME) 13 September 2000 (2000-09-13) the whole document ---	1-5,8-11 6,7,12, 13
Y	EP 0 541 216 A (INT COMPUTERS LTD) 12 May 1993 (1993-05-12) the whole document ---	6,7,12, 13
A	US 5 974 531 A (MA RUEY-LIANG ET AL) 26 October 1999 (1999-10-26) the whole document ---	1,8

-/--

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
25 June 2001	04/07/2001

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Klocke, L

2

INTERNATIONAL SEARCH REPORT

International Application No
PCT/L_ 01/04743

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	EP 0 949 564 A (MATSUSHITA ELECTRIC IND CO LTD) 13 October 1999 (1999-10-13) the whole document	
A	ANDREWS K ET AL: "MIGRATING A CISC COMPUTER FAMILY ONTO RISC VIA OBJECT CODE TRANSLATION" ACM SIGPLAN NOTICES, US, ASSOCIATION FOR COMPUTING MACHINERY, NEW YORK, vol. 27, no. 9, 1 September 1992 (1992-09-01), pages 213-222, XP000330602 ISSN: 0362-1340 the whole document	
A	O'CONNOR J -M ET AL: "PICOJAVA -I: THE JAVA VIRTUAL MACHINE IN HARDWARE" IEEE MICRO, US, IEEE INC. NEW YORK, vol. 17, no. 2, 1 March 1997 (1997-03-01), pages 45-53, XP000686468 ISSN: 0272-1732	

2

INTERNATIONAL SEARCH REPORT

In ation on patent family members

International Application No
PCT/L_ 01/04743

Patent document cited in search report		Publication date	Patent family member(s)			Publication date
WO 9927439	A	03-06-1999	AU	1174799	A	15-06-1999
			CN	1279782	T	10-01-2001
			EP	1035471	A	13-09-2000
EP 0541216	A	12-05-1993	AU	2744992	A	06-05-1993
			DE	69225195	D	28-05-1998
			DE	69225195	T	24-09-1998
			JP	5224921	A	03-09-1993
			US	5530816	A	25-06-1996
			ZA	9206505	A	04-03-1993
US 5974531	A	26-10-1999	NONE			
EP 0949564	A	13-10-1999	JP	11296381	A	29-10-1999